

Esercizi di Java Avanzato

Università di Napoli “Federico II”
Corso di Laurea in Informatica
Corso di Linguaggi di Programmazione II
Archivio completo degli esercizi d'esame

Marco Faella

4 aprile 2022

Quest'opera è offerta con licenza Creative Commons Attribuzione-NonCommerciale 4.0 Internazionale. Per visionare il testo della licenza, visitare <https://creativecommons.org/licenses/by-nc/4.0/legalcode.it>.

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Autore: Marco Faella, marfaella@gmail.com



Indice

| | |
|---|------------|
| 1 Uguaglianza tra oggetti | 1 |
| 2 Binding dinamico | 9 |
| 3 Esercizi elementari | 53 |
| 4 Java Collection Framework (collezioni) | 69 |
| 5 Scelta della firma | 93 |
| 6 Trova l'errore | 103 |
| 7 Design by contract | 107 |
| 8 Programmazione parametrica (generics) | 109 |
| 9 Classe mancante | 119 |
| 10 Classi interne | 125 |
| 11 Classi enumerate | 133 |
| 12 Vero o falso | 137 |
| 13 Lambda-espressioni | 163 |
| 14 Clonazione di oggetti | 165 |
| 15 Riflessione | 169 |
| 16 Multi-threading | 171 |
| 17 Iteratori e ciclo for-each | 195 |
| 18 Criteri di ordinamento tra oggetti | 199 |
| 19 Indice Cronologico | 211 |

1 Uguaglianza tra oggetti

1. (2022-1-26)

In riferimento all'esercizio 111, dire quali dei seguenti sono criteri di uguaglianza validi tra oggetti di tipo **Exchange**. In caso negativo, dire quale o quali proprietà sono violate e descrivere un controsenso. Due oggetti **Exchange** sono uguali se:

- a) Contengono gli stessi titoli, anche con valori diversi.
- b) Tutti i titoli in comune hanno lo stesso valore (ma potrebbero avere anche titoli diversi).
- c) Almeno uno dei due **Exchange** ha un titolo con almeno 2 alert.

2. (Uguaglianza tra cart, 2020-1-24)

Con riferimento alla classe **Cart** dell'esercizio 1, dire quali delle seguenti sono specifiche valide per il metodo **equals**. In caso negativo, motivare la risposta con un controsenso.

Due istanze di **Cart** sono uguali se:

- a) entrambi i carrelli contengono almeno un prodotto e il prodotto più caro del primo carrello ha lo stesso prezzo del prodotto più caro del secondo carrello;
- b) entrambi i carrelli sono vuoti, oppure entrambi contengono almeno un prodotto;
- c) i due carrelli contengono almeno un prodotto in comune;
- d) i prezzi totali di ciascun carrello differiscono al più di 10 euro.

3. (Student, 2019-7-23)

La classe **Student** è caratterizzata dai seguenti campi:

```
private String nome;  
private long matricola;
```

Dire quali dei seguenti sono criteri di uguaglianza validi per tale classe, giustificando la risposta.
Due studenti **a** e **b** sono uguali se:

- a) Hanno lo stesso nome oppure la stessa matricola.
- b) Hanno la matricola della stessa parità (entrambe pari o entrambe dispari).
- c) Hanno lo stesso nome, ma matricole diverse.
- d) Hanno entrambi i nomi **null**, oppure nessuno dei due nomi è **null** e hanno la stessa matricola.

Per ognuno dei criteri validi, implementare il metodo **hashCode** in modo coerente con **equals**.

4. (Box e ColoredBox, 2019-6-24)

Considerare le seguenti classi:

```
public class Box {  
    private int x, y;  
    ...  
}
```

```
public class ColoredBox extends Box {  
    private String color;  
    ...  
}
```

Sovrascrivere i metodi **equals** e **hashCode** rispettando le seguenti specifiche:

- Due **Box** sono uguali se hanno le stesse dimensioni **x** e **y**.
- Due **ColoredBox** sono uguali se, in aggiunta, hanno anche lo stesso colore.
- Un **Box** non è mai uguale a un **ColoredBox**.
- **hashCode** deve essere coerente con **equals**.

1 Uguaglianza tra oggetti

Precisare in quale/i classe/i è necessario sovrascrivere i metodi per ottenere questo comportamento.

5. (Fraction, 2018-7-19)

Implementare la classe **Fraction**, che rappresenta una frazione, e la sottoclasse **ReducedFraction**, che rappresenta una frazione irriducibile.

Due oggetti di questi tipi devono essere uguali per **equals** se rappresentano lo stesso numero razionale, anche se uno è di tipo **Fraction** e l'altro **ReducedFraction**.

Oltre a un costruttore che accetta numeratore e denominatore, le due classi offrono il metodo **times**, che calcola il prodotto, restituendo un nuovo oggetto **Fraction**. Il nuovo oggetto deve essere di tipo effettivo **ReducedFraction** se e soltanto se entrambi gli operandi del prodotto sono di tipo effettivo **ReducedFraction**.

Suggerimento: per calcolare il massimo comun divisore tra due interi **a** e **b**, si può usare l'istruzione **BigInteger.valueOf(a).gcd(BigInteger.valueOf(b)).intValue()**.

| Esempio d'uso: | Output: |
|--|--|
| <pre>Fraction a = new Fraction(12,30), b = new ReducedFraction (12,30), c = new Fraction(1,4), d = c.times(a); System.out.println(a); System.out.println(b); System.out.println(d); System.out.println(a.equals(b)); System.out.println(c.times(b));</pre> | 12/30 2/5 12/120 true 2/20 |

6. (Studente, 2018-3-23)

Implementare la classe **Studente** e le due sottoclassi **Triennale** e **Magistrale**. Uno studente è caratterizzato da nome e matricola. Ciascuna sottoclasse ha un prefisso che viene aggiunto a tutte le sue matricole. Due studenti sono considerati uguali da **equals** se hanno lo stesso nome e la stessa matricola (compreso il prefisso).

L'implementazione deve rispettare il seguente esempio d'uso:

| Esempio d'uso: | Output: |
|--|----------------------------------|
| <pre>Studente.Triennale.setPrefisso ("N86"); Studente.Magistrale.setPrefisso ("N97"); Object luca1 = new Studente.Triennale("Luca", "004312"), luca2 = new Studente.Triennale("Luca", "004312"), anna1 = new Studente.Triennale("Anna", "004312"), anna2 = new Studente.Magistrale("Anna", "004312"); System.out.println(luca1.equals(luca2)); System.out.println(anna1.equals(anna2)); System.out.println(anna1);</pre> | true false Anna N86/004312 |

7. (Room equals, 2017-4-26)

Con riferimento alla classe **Room** dell'esercizio 2, dire quali delle seguenti sono specifiche valide per il metodo **equals**, motivando la risposta.

Due istanze di **Room** sono uguali se:

- esiste una data in cui entrambe le camere sono prenotate;
- entrambe le camere non hanno alcuna prenotazione;
- entrambe le camere non hanno alcuna prenotazione, oppure esiste una persona che ha prenotato entrambe le camere (anche in date diverse);
- le camere sono state prenotate negli stessi giorni (anche da persone diverse).

8. (Polygon, 2017-2-23)

La classe `Polygon`, di cui si riporta un frammento, rappresenta un poligono nel piano cartesiano.

```
public class Polygon {  
    private static class Vertex {  
        double x, y;  
    }  
    private List<Vertex> vertices;  
    ...  
}
```

Dire quali delle seguenti sono specifiche valide per l'uguaglianza tra oggetti di tipo `Polygon`, giustificando la risposta.

Due poligoni *a* e *b* sono uguali se:

- a) Hanno lo stesso numero di vertici
- b) Sono entrambi triangoli
- c) Hanno gli stessi vertici, anche se in ordine diverso
- d) Hanno almeno un vertice in comune
- e) Si trovano nello stesso quadrante (con tutti i loro vertici)

Implementare la specifica (d) come metodo `equals` di `Polygon`.

9. (Engine, 2016-4-21)

[CROWDGRADER] Realizzare la classe `Engine`, che rappresenta un motore a combustione, caratterizzato da cilindrata (in cm³) e potenza (in cavalli). Normalmente, due oggetti `Engine` sono uguali se hanno la stessa cilindrata e la stessa potenza. Il metodo `byVolume` converte questo `Engine` in modo che venga confrontata solo la cilindrata. Analogamente, il metodo `byPower` converte questo `Engine` in modo che venga confrontata solo la potenza.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Engine a = new Engine(1200, 69), b = new Engine(1200, 75), c =  
    new Engine(1400, 75);  
System.out.println(a);  
System.out.println(a.equals(b));  
  
Engine aVol = a.byVolume(), bVol = b.byVolume();  
System.out.println(aVol);  
System.out.println(aVol.equals(bVol));  
System.out.println(a==aVol);  
  
Engine bPow = b.byPower(), cPow = c.byPower();  
System.out.println(bPow);  
System.out.println(bPow.equals(cPow));
```

Output:

```
(1200.0 cm3, 69.0 CV)  
false  
(1200.0 cm3, 69.0 CV)  
true  
false  
(1200.0 cm3, 75.0 CV)  
true
```

10. (Soldier, 2016-3-3)

In un gioco, `Soldier` è una sottoclasse di `Unit`. Gli oggetti `Unit` sono dotati di un campo `health` (intero), mentre gli oggetti `Soldier` hanno anche un campo `name` (stringa).

Dire quali delle seguenti sono specifiche valide per l'uguaglianza tra oggetti di queste due classi, giustificando la risposta. (Quando si dice "un X" si intende "un oggetto di tipo effettivo X")

- a) Due `Unit` o due `Soldier` sono uguali se hanno lo stesso `health`. Uno `Unit` non è mai uguale ad un `Soldier`.
- b) Due `Unit` sono uguali se hanno lo stesso `health`. Due `Soldier` sono uguali se hanno lo stesso `health` e `name`. Uno `Unit` non è mai uguale ad un `Soldier`.
- c) Due `Unit` o due `Soldier` sono uguali se hanno `health` maggiore di zero. Uno `Unit` non è mai uguale ad un `Soldier`.

1 Uguaglianza tra oggetti

- d) Due Unit sono uguali se hanno lo stesso health. Due Soldier sono uguali se hanno lo stesso name. Uno Unit è uguale ad un Soldier se hanno lo stesso health.

11. (Shape, 2014-4-28)

[CROWDGRADER] Per un programma di geometria piana, realizzare la classe astratta Shape e la sottoclasse concreta Circle. La classe Shape dispone dei metodi width, height, posX e posY, che restituiscono rispettivamente la larghezza, l'altezza, la posizione sulle ascisse e la posizione sulle ordinate del più piccolo rettangolo che contiene interamente la figura in questione (le coordinate restituite da posX e posY si riferiscono all'angolo in basso a sinistra del rettangolo).

Il costruttore di Circle accetta le coordinate del centro e il raggio, mentre il metodo setRadius consente di modificare il raggio.

Inoltre, le classi offrono le seguenti funzionalità:

- Gli oggetti Circle sono uguali secondo equals se hanno lo stesso centro e lo stesso raggio.
- Gli oggetti Shape sono clonabili.
- Gli oggetti Shape sono dotati di ordinamento naturale, sulla base dell'area del rettangolo che contiene la figura.

| Esempio d'uso: | Output: |
|--|--|
| <pre>Shape c1 = new Circle(2.0, 3.0, 1.0); Shape c2 = c1.clone(); System.out.println(c1.posX() + ", " + c1.posY()); System.out.println(c1.width() + ", " + c1.height()); System.out.println(c1.equals(c2)); ((Circle) c2).setRadius(2.0); System.out.println(c2.posX() + ", " + c2.posY());</pre> | <pre>1.0, 2.0 2.0, 2.0 true 0.0, 1.0</pre> |

12. (Shape equals, 2014-4-28)

Con riferimento all'esercizio 11, dire quali dei seguenti criteri è una valida specifica per l'uguaglianza tra oggetti di tipo Circle. In caso negativo, giustificare la risposta con un controesempio.

Due oggetti Circle sono uguali se:

- hanno lo stesso centro oppure lo stesso raggio;
- entrambe le circonferenze contengono l'origine all'interno oppure nessuna delle due la contiene;
- il raggio di uno dei due oggetti è maggiore di quello dell'altro.

13. (PeriodicTask, 2014-3-5)

Realizzare la classe PeriodicTask, che consente di eseguire un Runnable periodicamente, ad intervalli specificati. Il costruttore accetta un oggetto Runnable r e un numero di millisecondi p, detto periodo, e fa partire un thread che esegue r.run() ogni p millisecondi (si noti che il costruttore non è bloccante). Il metodo getTotalTime restituisce il numero complessivo di millisecondi che tutte le chiamate a r.run() hanno utilizzato fino a quel momento.

Suggerimento: il seguente metodo della classe System restituisce il numero di millisecondi trascorsi dal primo gennaio 1970: `public static long currentTimeMillis()`.

(15 punti) Inoltre, dire quali dei seguenti criteri di uguaglianza per oggetti di tipo PeriodicTask sono validi, giustificando brevemente la risposta. Due oggetti di tipo PeriodicTask sono uguali se:

- hanno lo stesso Runnable ed un periodo inferiore ad un secondo;
- hanno due periodi che sono l'uno un multiplo intero dell'altro (ad es. 5000 millisecondi e 2500 millisecondi);
- hanno lo stesso Runnable oppure lo stesso periodo.

| | |
|---|---|
| <p>Esempio d'uso:</p> <pre>Runnable r = new Runnable() { public void run() { System.out.println("Ciao!"); } }; new PeriodicTask(r, 2000);</pre> | <p>Output:</p> <pre>Ciao! Ciao! (dopo 2 secondi) Ciao! (dopo altri 2 secondi) ...</pre> |
|---|---|

14. (2013-4-29)

Con riferimento alla classe `Pair` dell'esercizio 2, dire quali delle seguenti specifiche per il metodo `equals` sono valide e perché.

Due istanze *a* e *b* di `Pair` sono uguali se...

- a) ...hanno almeno una delle due componenti uguale.
- b) ...hanno una delle due componenti uguale e una diversa.
- c) ...la prima componente di *a* è uguale alla seconda componente di *b* e la seconda componente di *a* è uguale alla prima componente di *b*.

15. (Cane, 2013-3-22)

Data la seguente classe:

```
public class Cane {
    private Person padrone;
    private String nome;
    ...
}
```

Si considerino le seguenti specifiche alternative per il metodo `equals`. Due oggetti *x* e *y* di tipo `Cane` sono uguali se:

- a) *x* e *y* non hanno padrone (cioè, `padrone == null`) e hanno lo stesso nome;
 - b) *x* e *y* hanno lo stesso padrone oppure lo stesso nome;
 - c) *x* e *y* hanno due nomi di pari lunghezza (vale anche `null` o stringa vuota);
 - d) *x* e *y* non hanno padrone **oppure** entrambi hanno un padrone ed i loro nomi iniziano con la stessa lettera (vale anche `null` o stringa vuota).
- Dire quali specifiche sono valide, fornendo un controsenso in caso negativo. (*16 punti*)
 - Implementare la specifica (c). (*5 punti*)

16. (MultiSet, 2013-2-11)

Un MultiSet è un insieme in cui ogni elemento può comparire più volte. Quindi, ammette duplicati come una lista, ma, a differenza di una lista, l'ordine in cui gli elementi vengono inseriti non è rilevante. Implementare una classe parametrica `MultiSet`, con i seguenti metodi:

- `add`, che aggiunge un elemento,
- `remove`, che rimuove un elemento (se presente), ed
- `equals`, che sovrascrive quello di `Object` e considera uguali due `MultiSet` se contengono gli stessi elementi, ripetuti lo stesso numero di volte.

Infine, deve essere possibile iterare su tutti gli elementi di un `MultiSet` usando il ciclo `for-each`.

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre>MultiSet<Integer> s1 = new MultiSet<Integer>(); MultiSet<Integer> s2 = new MultiSet<Integer>(); s1.add(5); s1.add(7); s1.add(5); s2.add(5); s2.add(5); s2.add(7); for (Integer n: s1) System.out.println(n); System.out.println(s1.equals(s2));</pre> | <p>Output (l'ordine dei numeri è irrilevante):</p> <pre>7 5 5 true</pre> |
|--|--|

17. (Insieme di lettere, 2013-1-22)

La classe `MyString` rappresenta una stringa. Due oggetti di tipo `MyString` sono considerati uguali (da `equals`) se utilizzano le stesse lettere, anche se in numero diverso. Ad esempio, “casa” è uguale a “cassa” e diverso da “sa”; “culle” è uguale a “luce” e diverso da “alluce”. La classe `MyString` deve essere clonabile e deve offrire un’implementazione di `hashCode` coerente con `equals` e non banale (che non restituisca lo stesso codice hash per tutti gli oggetti).

Suggerimento: Nella classe `String` è presente il metodo `public char charAt(int i)`, che restituisce l’i-esimo carattere della stringa, per i compreso tra 0 e `length()-1`.

Esempio d’uso:

```
MyString a = new MyString("freddo");
MyString b = new MyString("defro");
MyString c = new MyString("caldo");
MyString d = c.clone();
System.out.println(a.equals(b));
System.out.println(b.equals(c));
System.out.println(a.hashCode()==b.hashCode());
```

Output dell’esempio d’uso:
true
false
true

18. (Anagrammi, 2012-9-3)

Implementare la classe `MyString`, che rappresenta una stringa con la seguente caratteristica: due oggetti `MyString` sono considerati uguali (da `equals`) se sono uno l’anagramma dell’altro. Inoltre, la classe `MyString` deve essere clonabile e deve offrire un’implementazione di `hashCode` coerente con `equals` e non banale (che non restituisca lo stesso codice hash per tutti gli oggetti).

Suggerimento: Nella classe `String` è presente il metodo `public char charAt(int i)`, che restituisce l’i-esimo carattere della stringa, per i compreso tra 0 e `length()-1`.

Esempio d’uso:

```
MyString a = new MyString("uno_due_tre");
MyString b = new MyString("uno_tre_deu");
MyString c = new MyString("ert_unodue");
MyString d = c.clone();
System.out.println(a.equals(b));
System.out.println(b.equals(c));
System.out.println(a.hashCode()==b.hashCode());
```

Output dell’esempio d’uso:
true
false
true

19. (2012-4-23)

Con riferimento alla classe `Safe` dell’esercizio 2, dire quali delle seguenti specifiche per il metodo `equals` sono valide e perché.

Due istanze di `Safe` sono uguali se...

- ...hanno il messaggio segreto di pari lunghezza.
- ...hanno la stessa combinazione oppure lo stesso messaggio segreto.
- ...hanno la combinazione maggiore di zero.
- ...la somma delle due combinazioni è un numero pari.
- ...almeno uno dei due messaggi segreti contiene la parola “maggior domo”.

20. (Operai, 2011-2-7)

Un operaio (classe `Op`) è caratterizzato da nome (`String`) e salario (`short`), mentre un operaio specializzato (classe `OpSp`, sottoclasse di `Op`), in aggiunta possiede una specializzazione (riferimento ad un oggetto di tipo `Specialty`). Dire quali dei seguenti sono criteri validi di uguaglianza (`equals`) tra operai e operai specializzati, giustificando la risposta.

Implementare comunque il criterio (a), indicando chiaramente in quale/i classe/i va ridefinito il metodo `equals`.

- Due operai semplici sono uguali se hanno lo stesso nome. Due operai specializzati, in più, devono avere anche la stessa specializzazione (nel senso di `==`). Un operaio semplice non è mai uguale ad un operaio specializzato.

- b) Come il criterio (a), tranne che un operaio semplice è uguale ad un operaio specializzato se hanno lo stesso nome e la specializzazione di quest'ultimo è `null`.
- c) Due operai di qualunque tipo sono uguali se hanno lo stesso salario.
- d) Gli operai semplici sono tutti uguali tra loro. Ciascun operaio specializzato è uguale solo a se stesso.

21. (2008-7-9)

Data la seguente classe.

```
public class Z {
    private Z other;
    private int val;
    ...
}
```

Si considerino le seguenti specifiche alternative per il metodo `equals`. Due oggetti `x` e `y` di tipo `Z` sono uguali se:

- a) `x.other` e `y.other` puntano allo stesso oggetto ed `x.val` è maggiore o uguale di `y.val`;
 - b) `x.other` e `y.other` puntano allo stesso oggetto ed `x.val` e `y.val` sono entrambi pari;
 - c) `x.other` e `y.other` puntano allo stesso oggetto oppure `x.val` è uguale a `y.val`;
 - d) `x.other` e `y.other` sono entrambi `null` oppure nessuno dei due è `null` ed `x.other.val` è uguale a `y.other.val`.
- Dire quali specifiche sono valide e perché. (*20 punti*)
 - Implementare la specifica (d). (*10 punti*)

22. (2008-4-21)

Con riferimento all'Esercizio 1, ridefinire il metodo `equals` per i triangoli, in modo da considerare uguali i triangoli che hanno lati uguali. Dire esplicitamente in quale classe (o quali classi) va ridefinito il metodo.

23. (Monomio, 2007-2-7)

Un *monomio* è una espressione algebrica del tipo $a_n \cdot x^n$, cioè è un particolare tipo di polinomio composto da un solo termine. Implementare una classe `Monomial` come sottoclasse di `Polynomial`. La classe `Monomial` deve offrire un costruttore che accetta il grado n e il coefficiente a_n che identificano il monomio.

Ridefinire il metodo `equals` in modo che si possano confrontare liberamente polinomi e monomi, con l'ovvio significato matematico di egualanza.

| Esempio d'uso: | Output dell'esempio d'uso: |
|--|---|
| <pre>double a1[] = {1, 2, 3}; double a2[] = {0, 0, 0, 5}; Polynomial p1 = new Polynomial(a1); Polynomial p2 = new Polynomial(a2); Polynomial p3 = new Monomial(3, 5); System.out.println(p2); System.out.println(p3); System.out.println(p3.equals(p1)); System.out.println(p3.equals(p2)); System.out.println(p2.equals(p3)); System.out.println(p2.equals((Object) p3));</pre> | <pre>5.0 x^3 5.0 x^3 false true true true</pre> |

2 Binding dinamico

24. (2019-7-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object x, B y) { return "A1"; }  
    private String f(B x, B y) { return "A2" + f(x, y); }  
    public String f(A x, Object y) { return "A3"; }  
}  
class B extends A {  
    private String f(B x, B y) { return "B1" + f(null, (A)y); }  
    public String f(Object x, B y) { return "B2" + f(x, (A)y); }  
    public String f(A x, B y) { return "B3" + f(y, y); }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new B();  
        A alfa = beta;  
        System.out.println(alfa.f(null, alfa));  
        System.out.println(beta.f(beta, beta));  
        System.out.println(beta.getClass() == alfa.getClass());  
    }  
}
```

- Per ogni chiamata ad un metodo `f`, indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

25. (2019-6-24)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object x, B y) { return "A1"; }  
    public String f(B x, B y) { return "A2" + f(x, y); }  
    public String f(A x, Object y) { return "A3"; }  
}  
class B extends A {  
    public String f(B x, B y) { return "B1" + f(null, (A)y); }  
    private String f(A x, B y) { return "B2" + f(y, y); }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new B();  
        A alfa = beta;  
        System.out.println(alfa.f(beta, null));  
        System.out.println(beta.f(beta, beta));  
        System.out.println(beta.getClass() == alfa.getClass());  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

26. (2019-4-29)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object a, B b) { return "A1"; }
    public String f(A a, A b) { return "A2"; }
    public String f(B a, C b) { return "A3"; }
}
class B extends A {
    public String f(Object a, A b) { return "B1" + f(null, new B()); }
    private String f(A a, B b) { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1"; }
    public String f(A a, B b) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(alfa.f(beta, gamma));
        System.out.println(beta.f(beta, beta));
        System.out.println(gamma.f(alfa, null));
        System.out.println(beta instanceof A);
    }
}

```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

27. (2019-3-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A[] y) { return "A1"; }
    public String f(A x, Object y) { return "A2:" + x.f(new B(), null); }
}
class B extends A {
    public String f(B x, B[] y) { return "B1"; }
    public String f(A x, A[] y) { return "B2"; }
    public String f(A x, Object[] y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B[] arrayB = new B[10];
        A[] arrayA = arrayB;
        arrayB[0] = new B();
        System.out.println(arrayB[0].f(null, arrayB));
        System.out.println(arrayA[0].f(null, arrayA));
        System.out.println(arrayA[0].f(arrayA[0], null));
    }
}

```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

28. (2019-2-15)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(A x, A[] y) { return "A1"; }  
    public String f(A x, Object y) { return "A2:" + x.f(new C(), y); }  
}  
class B extends A {  
    public String f(C x, A[] y) { return "B1:" + x.f((A)x, y); }  
    public String f(A x, A[] y) { return "B2"; }  
    public String f(A x, Object[] y) { return "B3"; }  
}  
class C extends B {  
    public String f(A x, B[] y) { return "C1"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = gamma;  
        B[] array = new B[10];  
        System.out.println(beta.f(gamma, array));  
        System.out.println(gamma.f(beta, null));  
        System.out.println(beta.f(array[0], null));  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

29. (2019-1-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(A x, A y, B z) { return "A1"; }  
    public String f(A x, Object y, A z) { return "A2"; }  
    private String f(B x, Object y, B z) { return "A3"; }  
}  
class B extends A {  
    public String f(A x, A y, B z) { return "B1" + f(x, this, z); }  
    private String f(A x, B y, B z) { return "B2"; }  
    public String f(B x, Object y, B z) { return "B3"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new B();  
        A alfa = beta;  
  
        System.out.println(alfa.f(alfa, alfa, null));  
        System.out.println(beta.f(alfa, beta, alfa));  
        System.out.println(beta.f(beta, beta, beta));  
        System.out.println(beta.f(alfa, alfa, null));  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

2 Binding dinamico

30. (2018-9-17)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object x, A y) { return "A1"; }  
    private String f(A x, Object y) { return "A2"; }  
    protected String f(A x, B y) { return "A3"; }  
}  
class B extends A {  
    public String f(B x, B y) { return "B1" + f(x, (Object)y); }  
    public String f(A x, Object y) { return "B2"; }  
}  
class C extends B {  
    public String f(A x, Object y) { return "C1" + f(x, (B)y); }  
    public String f(Object x, A y) { return "C2"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new C();  
        A alfa = beta;  
        System.out.println(alfa.f(beta, null));  
        System.out.println(beta.f(beta, beta));  
        System.out.println(beta.f(alfa, (B)null));  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

31. (2018-7-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object a, A b) { return "A1"; }  
    public String f(A a, B b) { return "A2"; }  
}  
class B extends A {  
    public String f(B a, B b) { return "B1" + f(a, (A)b); }  
    public String f(A a, B b) { return "B2"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new B();  
        A alfa = beta;  
        System.out.println(alfa.f(beta, null));  
        System.out.println(beta.f(beta, beta));  
        System.out.println(beta.f(alfa, null));  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

32. (2018-6-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, A y, B z) { return "A1"; }
    public String f(A x, C y, C z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, A z) { return "B1" + f(null, new B(), y); }
    private String f(A x, B y, B z) { return "B2"; }
}
class C extends B {
    public String f(A x, A y, B z) { return "C1"; }
    public String f(A x, C y, C z) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(alfa.f(beta, gamma, gamma));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(gamma.f(alfa, null, beta));
    }
}

```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

33. (2018-5-2)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object a, A b) { return "A1"; }
    public String f(A a, C b) { return "A2"; }
}
class B extends A {
    public String f(Object a, A b) { return "B1" + f(null, new B()); }
    private String f(A a, B b) { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1"; }
    public String f(A a, B b) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(alfa.f(beta, gamma));
        System.out.println(beta.f(beta, beta));
        System.out.println(gamma.f(alfa, null));
        System.out.println(beta instanceof A);
    }
}

```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

34. (2018-3-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, A y, B z) { return "A1"; }
    private String f(A x, B y, A z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, B z) { return "B1" + f(null, z, new B()); }
    private String f(B x, B y, B z) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = (A) beta;

        System.out.println(alfa.f(alfa, beta, beta));
        System.out.println(alfa.f(beta, alfa, null));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(alfa instanceof B);
    }
}

```

- Per ogni chiamata ad un metodo (escluso `println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

35. (2018-10-18)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}
class B extends A {
    public String f(A a, B b) { return "2"; }
    public String f(C c, B b) { return "3"; }
    public int f(C c, Object x) { return 4; }
}
class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(null, gamma));
        System.out.println(beta.f(gamma, gamma));
        System.out.println(gamma.f(gamma, alfa));
        System.out.println(gamma.f(beta, beta));
        System.out.println(1 + "1");
    }
}

```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

36. (2017-7-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object x, A y, A z) { return "A1"; }  
    private String f(A x, B y, B z) { return "A2"; }  
}  
class B extends A {  
    public String f(Object x, A y, A z) { return "B1" + f(null, new B(), new C()); }  
    private String f(B x, B y, C z) { return "B2"; }  
}  
class C extends B {  
    public String f(A x, B y, B z) { return "C1" + f(this, this, z); }  
    public String f(B x, B y, B z) { return "C2"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = gamma;  
        A alfa = gamma;  
  
        System.out.println(beta.f(beta, beta, gamma));  
        System.out.println(gamma.f(beta, null, beta));  
        System.out.println(gamma.f(alfa, beta, gamma));  
        System.out.println( ((Object)gamma).equals( (Object)alfa ) );  
    }  
}
```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (esclusi `println` ed `equals`) indicare la lista delle firme candidate.

37. (2017-6-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object x, A y, A z) { return "A1"; }  
    private String f(A x, B y, B z) { return "A2"; }  
}  
class B extends A {  
    public String f(A x, B y, B z) { return "B1" + f(null, new B(), z); }  
    private String f(B x, B y, C z) { return "B2"; }  
}  
class C extends B {  
    public String f(A x, B y, B z) { return "C1" + f(this, this, z); }  
    public String f(B x, B y, B z) { return "C2"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = gamma;  
        A alfa = gamma;  
  
        System.out.println(alfa.f(alfa, beta, gamma));  
        System.out.println(gamma.f(beta, beta, beta));  
        System.out.println(gamma.f(alfa, beta, null));  
        System.out.println( ((Object)beta).equals(alfa) );  
    }  
}
```

- Indicare l'output del programma.

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

38. (2017-4-26)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, A y, A z) { return "A1"; }
    private String f(B x, C y, B z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, B z) { return "B1" + f(null, new B(), z); }
    public String f(A x, B y, C z) { return "B2"; }
}
class C extends B {
    public String f(Object x, A y, B z) { return "C1" + f(this, this, z); }
    public String f(B x, C y, B z) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(alfa.f(alfa, beta, gamma));
        System.out.println(gamma.f(beta, beta, beta));
        System.out.println(gamma.f(beta, beta, null));
        System.out.println(128 & 4);
    }
}
```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

39. (2017-3-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    String f(A x, int z) { return "A1:" + f(this, z/2.0); }
    public String f(A x, double z) { return "A2"; }
}
class B extends A {
    public String f(A x, float z) { return "B1"; }
    public String f(B x, int z) { return "B2"; }
}
class C extends B {
    public String f(A x, int z) { return "C1"; }
    public String f(B x, int z) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(gamma, 42));
        System.out.println(beta.f(null, 3.14));
        System.out.println(beta.f(beta, 7));
        System.out.println(16 & 32);
    }
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

40. (2017-2-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    private String f(A x, C z) { return "A1"; }
    public String f(A x, B z) { return "A2:" + x.f(this, (C)x); }
}
class B extends A {
    public String f(A x, C z) { return "B1"; }
    public String f(B x, B z) { return "B2"; }
}
class C extends B {
    public String f(B x, B z) { return "C1"; }
    public String f(B x, C z) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(gamma, null));
        System.out.println(beta.f(alfa, beta));
        System.out.println(beta.f(null, beta));
        System.out.println(alfa.getClass() == A.class);
    }
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

41. (2017-10-6)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, Object y, B z) { return "A1"; }
    private String f(A x, B y, B z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, A z) { return "B1" + f(null, new B(), new B()); }
    private String f(B x, B y, B z) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(alfa, beta, beta));
        System.out.println(beta.f(beta, alfa, alfa));
        System.out.println(beta.f(null, beta, beta));
        System.out.println(beta.equals((Object)alfa));
    }
}
```

- Indicare l'output del programma.

- Per ogni chiamata ad un metodo (esclusi `println` ed `equals`) indicare la lista delle firme candidate.

42. (2017-1-25)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] y, B z) { return "A1"; }
    public String f(A x, Object y, B z) { return "A2"; }
}
class B extends A {
    public String f(B x, A[] y, B z) { return "B1:" + x.f((A)x, y, z); }
    public String f(A x, B[] y, B z) { return "B2"; }
}
class C extends B {
    public String f(A x, A[] y, C z) { return "C1:" + z.f(new C(), y, z); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A[] array = new A[10];
        System.out.println(beta.f(gamma, array, gamma));
        System.out.println(gamma.f(array[0], null, beta));
        System.out.println(beta == gamma);
    }
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

43. (2016-9-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] y) { return "A1"; }
    public String f(A x, Object y) { return "A2:" + x.f(new C(), y); }
}
class B extends A {
    public String f(B x, A[] y) { return "B1:" + x.f((A)x, y); }
    public String f(A x, B[] y) { return "B2"; }
}
class C extends B {
    public String f(A x, A[] y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A[] array = new A[10];
        System.out.println(beta.f(gamma, array));
        System.out.println(gamma.f(beta, null));
    }
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

44. (2016-7-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(A x, A y) { return "A1"; }  
    private String f(A x, Object y) { return "A2"; }  
}  
class B extends A {  
    public String f(B x, A y) { return "B1:" + y.f(y, y); }  
    public String f(A x, A y) { return "B2"; }  
}  
class C extends B {  
    public String f(A x, A y) { return "C1:" + y.f(y, null); }  
}  
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = gamma;  
        A alfa = new A();  
        System.out.println(beta.f(gamma, alfa));  
        System.out.println(gamma.f(alfa, alfa));  
        System.out.println((12 & 3) > 0);  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

45. (2016-6-22)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(Object x, A y) { return "A0"; }  
    public String f(A[] x, A y) { return "A1"; }  
    public String f(A[] x, B y) { return "A2"; }  
}  
class B extends A {  
    public String f(A[] x, A y) { return "B1"; }  
    public String f(B x, A y) { return "B2:" + f((A) x, null); }  
    public String f(B[] x, A y) { return "B3"; }  
    private String f(A x, A y) { return "B4"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new B();  
        A alfa = beta;  
        B[] arr = new B[10];  
        System.out.println(alfa.f(null, alfa));  
        System.out.println(beta.f(arr[0], alfa));  
        System.out.println(beta.f(arr[0], arr[1]));  
        System.out.println(beta.f(arr, beta));  
        System.out.println(5871 & 5871);  
    }  
}
```

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.
- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.

46. (2016-4-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object a, A b) { return "A1"; }
    private String f(B a, C b) { return "A2"; }
}
class B extends A {
    public String f(Object a, A b) { return "B1" + a + b; }
    public String f(A a, B b) { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1" + a + b; }
    public String f(B a, C b) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(alfa.f(beta, gamma));
        System.out.println(gamma.f(beta, beta));
        System.out.println(gamma.f(beta, null));
        System.out.println(8 & 4);
    }
}
```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

47. (2016-3-3)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
abstract class A {
    public abstract String f(A[] x, Object y);
    public String f(Object[] x, A[] y) { return "A2"; }
}
class B extends A {
    public String f(B[] x, Object y) { return "B1"; }
    public String f(A[] x, Object y) { return "B2"; }
    public String f(B[] x, A[] y) { return "B3"; }
}
class C extends B {
    public String f(B[] x, A[] y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new C();
        A alfa = new B();
        B[] array = new B[10];
        System.out.println(alfa.f(array, beta));
        System.out.println(beta.f(array, beta));
        System.out.println(beta.f(array, array));
        System.out.println(beta.f(null, array));

        Object betaclass = beta.getClass();
        System.out.println(betaclass instanceof B);
        System.out.println(betaclass instanceof C);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

48. (2016-1-27)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, Object y) { return "A1"; }
    public String f(Object x, A y) { return "A2"; }
}
class B extends A {
    public String f(A x, Object y) { return "B1"; }
    private String f(A x, A y) { return "B2"; }
    public String f(B x, B y) { return "B3"; }
}
class C extends B {
    public String f(B x, B y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new C();
        A alfa = beta;
        System.out.println(beta.f((C)alfa, beta));
        System.out.println(beta.f(beta, null));
        System.out.println(beta.f((Object)beta, alfa));
        System.out.println(alfa.f(beta, beta));

        System.out.println(alfa.getClass() == C.class);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

49. (2015-9-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, B y) { return "A1"; }
    public String f(C x, Object y) { return "A2"; }
}
class B extends A {
    public String f(A x, B y) { return "B1"; }
    private String f(A x, A y) { return "B2"; }
    public String f(C x, B y) { return "B3"; }
}
class C extends B {
    public String f(C x, B y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = new C();
        System.out.println(beta.f((C)alfa, alfa));
        System.out.println(beta.f(beta, null));
        System.out.println(alfa.f(alfa, beta));
        System.out.println(alfa.f((C)alfa, beta));
    }
}
```

```

        System.out.println(alfa .getClass() == C.class);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

50. (2015-7-8)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, long l, float m) { return "A1"; }
    public String f(A x, byte l, int m) { return "A2"; }
    public String f(B x, short l, boolean m) { return "A3"; }
}
class B extends A {
    public String f(A y, long m, float p) { return "B1"; }
    public String f(A y, long m, long p) { return "B2"; }
    public String f(Object y, double m, float p) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa .f(alfa , (short)500, 1));
        System.out.println(beta.f(alfa , (short)500, 1));
        System.out.println(beta.f(beta, (short)500, 1));
        System.out.println(beta.f(beta, (byte)1, 1));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

51. (2015-6-24)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, A y, B z) { return "A1"; }
    public String f(A x, Object y, B z) { return "A2"; }
    public String f(B x, Object y, B z) { return "A3"; }
}
class B extends A {
    public String f(A x, A y, B z) { return "B1"; }
    public String f(A x, Object y, A z) { return "B2"; }
    public String f(A x, Object y, B z) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa .f(alfa , alfa , null));
        System.out.println(beta.f(alfa , beta, beta));
        System.out.println(beta.f(beta, beta, beta));
        System.out.println(beta.f(alfa , alfa , alfa));
    }
}

```

```
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

52. (2015-2-5)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A y, B z) { return "A1"; }
    public String f(A x, B y, C z) { return "A2"; }
}
class B extends A {
    public String f(Object x, A y, B z) { return "B1"; }
    public String f(B x, B y, B z) { return "B2"; }
}
class C extends B {
    public String f(A x, A y, B z) { return "C1"; }
    public String f(C x, B y, A z) { return "C2" + f(z, y, null); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(beta.f(alfa, beta, beta));
        System.out.println(beta.f(gamma, beta, beta));
        System.out.println(gamma.f(beta, alfa, beta));
        System.out.println(gamma.f(gamma, beta, beta));
        System.out.println(129573 & 129572);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

53. (2015-1-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(Object x, B y) { return "A1"; }
    public String f(A[] x, B y) { return "A2"; }
}
class B extends A {
    public String f(Object x, B y) { return "B1+" + f(y, null); }
    public String f(B[] x, C y) { return "B2"; }
}
class C extends B {
    public String f(A[] x, A y) { return "C1+" + f(null, y); }
    public String f(B[] x, C y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        B[] beta = new C[10];
        A[] alfa = beta;
```

```

        beta[0] = new C();
        System.out.println(beta[0].f(beta, beta[0]));
        System.out.println(beta[0].f(alfa, beta[2]));
        System.out.println(beta[0].f(alfa, alfa[0]));
        System.out.println(6 & 7);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

54. (2014-9-18)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, short n) { return "A1"; }
    public String f(A x, int n) { return "A2"; }
    private String f(B x, double n) { return "A3"; }
}
class B extends A {
    public String f(A x, double n) { return "B1:" + f(x, (int)n); }
    public String f(B x, double n) { return "B2"; }
    public String f(A x, int n) { return "B3"; }
}
class C extends B {
    public String f(A x, int n) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(gamma, (byte)2));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(gamma.f(alfa, (float)5));
        System.out.println(11 | 3);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

55. (2014-7-3)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, int n) { return "A1"; }
    public String f(A x, int n) { return "A2:" + n; }
    public String f(A x, double n) { return "A3:" + x.f(x, (int) n); }
    private String f(B x, int n) { return "A4"; }
}
class B extends A {
    public String f(A x, double n) { return "B1:" + n; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(A x, int n) { return "B3"; }
}

```

```

public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(null, 2L));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(12 & 2);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

56. (2014-7-28)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, double n) { return "A1"; }
    public String f(A x, int n) { return "A2"; }
    private String f(B x, int n) { return "A3"; }
}
class B extends A {
    private String f(A x, double n) { return "B1"; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(A x, long n) { return "B3"; }
}
class C extends B {
    public String f(A x, int n) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(gamma, 2L));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(gamma.f(beta, 5.0));
        System.out.println(11 & 3);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

57. (2014-4-28)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public static int x = 0;
    public A() { x++; }

    private int f(int a, double b) { return x; }
    public int f(int a, float b) { return x+10; }
    public int f(double a, double b) { return x+20; }
    public String toString() { return f(x, x) + ""; }
}
class B extends A {
}

```

```

public int f(int a, float b) { return x+30; }
public int f(int a, int b) { return x+40; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println(alfa1);
        System.out.println(alfa2);
        System.out.println(beta);
        System.out.println(alfa2.f(2, 3.0));
        System.out.println(beta.f(2, 3));
        System.out.println(beta.f((float) 4, 5));
        System.out.println(15 & 3);
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

58. (2014-3-5)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, A y) { return "A1"; }
    public String f(A x, A y) { return "A2"; }
    public String f(Object x, B y) { return "A3"; }
}
class B extends A {
    public String f(A x, A y) { return "B1"; }
    public String f(B x, A y) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(null, alfa));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.f(alfa.f(alfa, alfa), beta));
        System.out.println(5 & 7);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

59. (2014-11-3)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object a, B b) { return "A1"; }
    public String f(C a, B b) { return "A2"; }
}
class B extends A {
    public String f(Object a, B b) { return "B1+" + f(b, null); }
}

```

```

    public String f(A a, B b) { return "B2"; }
}
class C extends B {
    public String f(Object a, B b) { return "C1+" + f(this, b); }
    private String f(B a, B b) { return "C2"; }
}
public class Test0 {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;

        System.out.println(gamma.f(beta, beta));
        System.out.println(alfa.f(beta, gamma));
        System.out.println(9 & 12);
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

60. (2014-11-28)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(C x, short n) { return "A1"; }
    public String f(A x, int n) { return "A2"; }
    String f(B x, double n) { return "A3"; }
}
class B extends A {
    public String f(A x, double n) { return "B1:" + f(x, (int)n); }
    public String f(B x, double n) { return "B2"; }
    public String f(A x, int n) { return "B3"; }
}
class C extends B {
    public String f(A x, int n) { return "C1"; }
    public String f(B x, double n) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(beta, (byte)2));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(gamma.f(alfa, (float)5));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

61. (2014-1-31)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, A y) { return "A1"; }
    public String f(A[] x, A y) { return "A2"; }
}

```

```

    public String f(Object[] x, B y) { return "A3"; }
}
class B extends A {
    public String f(A[] x, A y) { return "B1"; }
    public String f(B[] x, A y) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arrA = new B[10];
        B[] arrB = new B[10];
        arrA[0] = arrB[0] = new B();
        System.out.println(arrA[0].f(null, arrA[0]));
        System.out.println(arrB[0].f(arrA, arrB[0]));
        System.out.println(arrB[0].f(arrB, arrA[0]));
        System.out.println("1" + 1);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

62. (2013-9-25)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, A y) { return "A0"; }
    public String f(A[] x, A y) { return "A1"; }
    public String f(B[] x, A y) { return "A2"; }
}
class B extends A {
    public String f(A[] x, A y) { return "B1"; }
    public String f(B x, A y) { return "B2:" + f((A) x, null); }
    public String f(B[] x, A y) { return "B3"; }
    private String f(A x, Object y) { return "B4"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        B[] arr = new B[10];
        System.out.println(alfa.f(null, alfa));
        System.out.println(beta.f(arr, alfa));
        System.out.println(beta.f(arr, beta));
        System.out.println(234 & 234);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

63. (2013-7-9)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, double n) { return "A0"; }
    public String f(A[] x, int n) { return "A1"; }
    public String f(B[] x, float n) { return "A2:" + f(x[0], (int) n); }
}

```

```

}
class B extends A {
    public String f(A[] x, int n) { return "B1:" + n; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(B[] x, float n) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        B[] arr = new B[10];
        System.out.println(alfa.f(null, 2L));
        System.out.println(beta.f(arr, 5.0));
        System.out.println(beta.f(arr, 2));
        System.out.println(11 ^ 11);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

64. (2013-6-25)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, short n) { return "A0"; }
    public String f(A x, int n) { return "A1:" + n; }
    public String f(A x, double n) { return "A2:" + f(x, (int) n); }
    private String f(B x, int n) { return "A3"; }
}
class B extends A {
    public String f(A x, int n) { return "B1:" + n; }
    public String f(B x, double n) { return "B2:" + f((A) x, 2); }
    public String f(A x, float n) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(null, 2L));
        System.out.println(beta.f(beta, 5.0));
        System.out.println(8 | 4);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

65. (2013-4-29)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, Object y) { return "A1"; }
    public String f(Object x, B y) { return "A2"; }
}
class B extends A {
    private String f(B x, A y) { return "B1"; }
}

```

```

    public String f(Object x, B y) { return "B2"; }
}
class C extends B {
    public String f(B x, A y) { return "C1"; }
    public String f(A x, Object y) { return "C2:" + f(null, x); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(beta.f(null, beta));
        System.out.println(alfa.f(gamma, alfa));
        System.out.println(gamma.f(beta, alfa));
        System.out.println(5 | 8);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

66. (2013-3-22)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}
class B extends A {
    public String f(A a, B b) { return "2"; }
    public int f(B c, C b) { return 3; }
    public int f(C c, Object x) { return 4; }
}
class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(beta, null));
        System.out.println(beta.f(beta, beta));
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(gamma, gamma));
        System.out.println(beta.getClass().getName());
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

67. (2013-2-11)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}
class B extends A {
    public String f(A a, B b) { return "2"; }
    public String f(C c, B b) { return "3"; }
    public int f(C c, Object x) { return 4; }
}
class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(null, gamma));
        System.out.println(beta.f(gamma, gamma));
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(beta, beta));
        System.out.println(1 + "1");
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

68. (2013-12-16)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(Object x, A y) { return "A1"; }
    public String f(A[] x, A y) { return "A2"; }
    public String f(Object[] x, B y) { return "A3"; }
}
class B extends A {
    public String f(A[] x, A y) { return "B1"; }
    public String f(B[] x, A y) { return "B2:" + f(x, null); }
    public String f(B[] x, B y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arrA = new B[20];
        B[] arrB = new B[10];
        arrA[0] = arrB[0] = new B();
        System.out.println(arrA[0].f(null, arrB[0]));
        System.out.println(arrB[0].f(arrA, arrB[0]));
        System.out.println(arrB[0].f(arrB, arrA[0]));
        System.out.println(3 | 4);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

69. (2013-1-22)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A y) { return "A1"; }
    private String f(A x, Object y) { return "A2"; }
}
class B extends A {
    public String f(B x, A y) { return "B1:" + y.f(y, y); }
    public String f(A x, A y) { return "B2"; }
}
class C extends B {
    public String f(A x, A y) { return "C1:" + y.f(y, null); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = new A();
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(alfa, alfa));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

70. (2012-9-3)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, B y) { return "A1"; }
    public String f(B x, C y) { return "A2"; }
}
class B extends A {
    public String f(B x, C y) { return "B1:" + f(x, x); }
    public String f(B x, B y) { return "B2"; }
}
class C extends B {
    public String f(A x, A y) { return "C1:" + y.f(y, null); }
    public String f(B x, B y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = beta;
        System.out.println(beta.f(gamma, beta));
        System.out.println(gamma.f(beta, gamma));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

71. (2012-7-9)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, B y) { return "A1"; }
    public String f(B x, C y) { return "A2"; }
}
class B extends A {
    public String f(B x, C y) { return "B1:" + f(x, x); }
    public String f(B x, B y) { return "B2"; }
}
class C extends B {
    public String f(A x, B y) { return "C1"; }
    public String f(B x, B y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = beta;
        System.out.println(beta.f(null, alfa));
        System.out.println(gamma.f(beta, gamma));
        System.out.println(alfa.f(gamma, beta));
        System.out.println((1 >> 1) < 0);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

72. (2012-6-18)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(A x, int n) { return "A1:" + n; }
    public String f(A x, double n) { return "A2:" + n; }
}
class B extends A {
    public String f(A x, int n) { return "B1:" + n; }
    public String f(B x, Object o) { return "B2"; }
}
class C extends B {
    public String f(A x, int n) { return "C1:" + n; }
    public String f(C x, double n) { return "C2:" + f(x, x); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = new B();
        System.out.println(beta.f(null, 7));
        System.out.println(alfa.f(gamma, 5));
        System.out.println(gamma.f(beta, 3.0));
        System.out.println((1 << 1) > 1);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

73. (2012-4-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] arr) { return "A1"; }
    public String f(Object x, Object[] y) { return "A2"; }
}
class B extends A {
    public String f(B x, Object[] y) { return "B1->" + f(y, y); }
}
class C extends B {
    public String f(A x, A[] arr) { return "C1"; }
    public String f(Object x, Object y) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arr = new B[10];
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(beta.f(null, arr));
        System.out.println(gamma.f(arr, alfa));
        System.out.println(gamma.f(alfa, arr));
        System.out.println(1 << 1);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

74. (2011-3-4)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, B y) { return "A1"; }
    public String f(C x, Object y) { return "A2"; }
}
class B extends A {
    public String f(A x, B y) { return f(x, x); }
    private String f(A x, A y) { return "B2"; }
    public String f(C x, B y) { return "B3"; }
}
class C extends B {
    public String f(C x, B y) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = new C();
        System.out.println(alfa.f(alfa, alfa));
        System.out.println(alfa.f(alfa, beta));
        System.out.println(alfa.f((C) alfa, beta));
        System.out.println(alfa.getClass() == C.class);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (10 punti)

- Per ogni chiamata ad un metodo *user-defined*, indicare la lista delle firme candidate. (Attenzione: le chiamate in questo esercizio sono quattro) (*16 punti*)

75. (2011-2-7)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(int x, double y) { return f(x, x); }
    public String f(double x, double y) { return "A2"; }
}
class B extends A {
    public String f(long x, double y) { return f(x, x); }
    private String f(long x, long y) { return "B2"; }
    public String f(int x, double y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(1, 2));
        System.out.println(beta.f(1.0, 2));
        System.out.println((2 == 2) && (null instanceof Object));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*10 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (Attenzione: le chiamate in questo esercizio sono quattro) (*16 punti*)

76. (2010-9-14)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] arr) { return "A1"; }
    public String f(Object x, B y) { return "A2"; }
}
class B extends A {
    public String f(B x, B y) { return "B1"; }
}
class C extends B {
    public String f(A x, A[] arr) { return "C1"; }
    public String f(B x, C y) { return "C2"; }
    private String f(C x, C y) { return "C3"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arr = new B[10];
        C gamma = new C();
        B beta = gamma;
        A alfa = null;
        System.out.println(beta.f(gamma, gamma));
        System.out.println(beta.f(beta, arr));
        System.out.println(gamma.f(beta, alfa));
        System.out.println(gamma.f(alfa, beta));
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*12 punti*)

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

77. (2010-7-26)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, int y) { return "A1"; }
    public String f(Object x, double y) { return "A2"; }
}
class B extends A {
    public String f(A x, int y) { return "B1"; }
}
class C extends B {
    public String f(B x, float y) { return "C1"; }
    public String f(Object x, double y) { return "C2"; }
    public String f(C x, int y) { return "C3:" + f(x, y * 2.0); }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(gamma, 3));
        System.out.println(gamma.f(null, 4));
        System.out.println(gamma.f(beta, 3));
        System.out.println("1" + 1);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

78. (2010-6-28)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    public String f(A x, A[] arr) { return "A1"; }
    public String f(Object x, Object y) { return "A2"; }
}
class B extends A {
    public String f(B x, Object[] y) { return "B1"; }
}
class C extends B {
    public String f(A x, A[] arr) { return "C1"; }
    public String f(B x, Object y) { return "C2"; }
    public String f(C x, B y) { return "C3"; }
}
public class Test {
    public static void main(String[] args) {
        A[] arr = new B[10];
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(beta.f(gamma, arr));
        System.out.println(gamma.f(arr, alfa));
        System.out.println(gamma.f(beta, alfa));
        System.out.println(5 | 7);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*16 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*9 punti*)

79. (2010-5-3)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {
    String f(A x, A y) { return "A1"; }
    String f(Object x, B y) { return "A2"; }
}
class B extends A {
    public String f(A x, A y) { return "B1"; }
}
class C extends B {
    public String f(B x, B y) { return "C1"; }
    public String f(B x, Object y) { return "C2"; }
    private String f(C x, B y) { return "C3"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(beta, alfa));
        System.out.println(beta.f(beta, gamma));
        System.out.println(gamma.f(null, gamma));
        System.out.println(1e100 + 1);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*16 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*9 punti*)

80. (2010-2-24)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A { public String f(int a, int b, float c) { return "A1"; }
          public String f(int a, double b, float c) { return "A2"; }
}
class B extends A {
    public String f(int a, int b, float c) { return "B1"; }
    private String f(double a, float b, int c) { return "B2"; }
    public String f(double a, int b, float c) { return "B3"; }
}
class C extends B {
    public String f(int a, int b, float c) { return "C1"; }
    public String f(double a, float b, int c) { return "C2"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(beta.f(1.0, 2, 3));
```

```

        System.out.println(gamma.f(1, 2, 3));
        System.out.println(gamma.f(1.0, 2, 3));
        System.out.println(beta instanceof A);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)

81. (2010-11-30)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public String f(B x, Object y) { return f(x, x); }
    public String f(B x, A y) { return "A2"; }
}
class B extends A {
    public String f(B x, Object y) { return f(x, x); }
    private String f(B x, B y) { return "B2"; }
    public String f(B x, A y) { return "B3"; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(beta, "ciao"));
        System.out.println(beta.f(beta, new A[10]));
        System.out.println((1 == 2) || (7 >= 7));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (10 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (Attenzione: le chiamate in questo esercizio sono quattro) (16 punti)

82. (2010-1-22)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(int[] a, int l) { return "A1"; }
          public String f(int[] a, double d) { return "A2"; }
          public String f(Object o, int l) { return "A3"; }
}
class B extends A {
    public String f(double[] a, double d) { return "B1"; }
}
class C extends B {
    public final String f(int[] a, int l) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        int[] x = new int[10];
        System.out.println(alfa.f(x, 10));
        System.out.println(beta.f(x, x[1]));
        System.out.println(gamma.f(null, 10));
    }
}

```

```

        System.out.println(gamma.f(x, 3.0));
        System.out.println(alfa instanceof C);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)

83. (2009-9-1'8)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(double n, A x) { return "A1"; }
          public String f(double n, B x) { return "A2"; }
          public String f(int n,     Object x) { return "A3"; }
}
class B extends A {
    public String f(double n, B x) { return "B1"; }
    public String f(float n,   Object y) { return "B2"; }
}
class C extends A {
    public final String f(int n, Object x) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta  = new B();
        A alfa  = beta;
        System.out.println(alfa.f(3, beta));
        System.out.println(alfa.f(3.0, beta));
        System.out.println(beta.f(3.0, alfa));
        System.out.println(gamma.f(3, gamma));
        System.out.println(false || alfa.equals(beta));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)

84. (2009-7-9)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(double n, A x, A y) { return "A1"; }
          public String f(double n, B x, B y) { return "A2"; }
          public String f(int n,     B x, B y) { return "A3"; }
}
class B extends A {
    public String f(int n, B x, B y) { return "B1:" + x.f(3.0,x,y); }
    public String f(float n, A x, Object y) { return "B2"; }
}
public class Test {
    public static void main(String[] args) {
        B beta  = new B();
        A alfa  = beta;
        System.out.println(alfa.f(3, alfa, beta));
        System.out.println(alfa.f(4, beta, beta));
        System.out.println(beta.f(3, alfa, (Object) alfa));
        System.out.println(true && (alfa instanceof B));
    }
}

```

```

    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

85. (2009-6-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    String f(A x, A y) { return "A1"; }
    String f(B x, B y) { return "A2"; }
}
class B extends A {
    String f(B x, B y) { return "B1:" + x.f(x,y); }
}
class C extends B {
    String f(B x, B y) { return "C1"; }
    String f(B x, Object y) { return "C2"; }
    String f(C x, Object y) { return "C3"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(alfa, beta));
        System.out.println(beta.f(gamma, beta));
        System.out.println(gamma.f(gamma, alfa));
        System.out.println(gamma.f(alfa, gamma));
        int x=0;
        System.out.println( (true || (x++>0)) + ":" + x);
    }
}
```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

86. (2009-4-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    String f(A other, int n) { return "A1:" + n; }
    String f(B other, long n) { return "A2:" + n; }
}
class B extends A {
    String f(A other, int n) { return "B1:" + n; }
}
class C extends B {
    String f(B other, long n) { return "C1:" + n; }
    String f(B other, double n) { return "C2:" + n; }
    private String f(C other, long n) { return "C3:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, 4L));
    }
}
```

```

        System.out.println(beta.f(gamma, 4L));
        System.out.println(gamma.f(null, 3L));
        System.out.println(7 >> 1);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*16 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*9 punti*)

87. (2009-2-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(double a, long b, long c) { return 1; }
    public int f(double a, int b, double c) { return 2; }
}
class B extends A {
    public int f(int a, double b, long c) { return 3; }
    public int f(int a, int b, double c) { return 4; }
    public int f(double a, int b, double c) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(alfa.f(1, 2, 3.0));
        System.out.println(beta.f(1, 2, 3.0));
        System.out.println(beta.f(1, 21, 3));
        System.out.println(762531 & 762531);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*15 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*12 punti*)

88. (2009-11-27)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A { public String f(double n, Object x) { return "A1"; }
          public String f(double n, A x) { return "A2"; }
          public String f(int n, Object x) { return "A3"; }
}
class B extends A {
    public String f(double n, Object x) { return "B1"; }
    public String f(float n, Object y) { return "B2"; }
}
class C extends B {
    public final String f(double n, A x) { return "C1"; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(3.0, gamma));
        System.out.println(beta.f(3, beta));
    }
}

```

```

        System.out.println(beta.f(3.0, null));
        System.out.println(gamma.f(3.0, gamma));
        System.out.println(true && (alfa==beta));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (12 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (15 punti)

89. (2009-1-29)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public int f(int i, long l1, long l2) { return 1; }
    public int f(int i1, int i2, double d) { return 2; }

}

class B extends A {
    public int f(boolean b, double d, long l) { return 3; }
    public int f(boolean b, int i, double d) { return 4; }
    public int f(int i1, int i2, double d) { return 5; }
}

public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;
        System.out.println(alfa.f(1, 2, 3));
        System.out.println(alfa.f(1, 2, 3.0));
        System.out.println(beta.f(true, 5, 6));
        System.out.println(beta.f(false, 3.0, 4));
        System.out.println(7 & 5);
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (15 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (12 punti)

90. (2009-1-15)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a, B b);
    public int f(B b, C c) { return 1; }
}

class B extends A {
    public String f(A a, B b) { return "2"; }
    public String f(C c, B b) { return "3"; }
    public int f(C c, Object x) { return 4; }
}

class C extends B {
    public String f(C c1, C c2) { return "5"; }
    public String f(A a, B b) { return "6"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
    }
}

```

```

        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, null));
        System.out.println(alfa.f(null, gamma));
        System.out.println(beta.f(gamma, alfa));
        System.out.println(gamma.f(beta, beta));
        System.out.println(1 + "1");
    }
}

```

- Indicare l'output del programma. (*15 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*12 punti*)

91. (2008-9-8)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int a, int b, float c) { return 1; }
    public int f(int a, double b, int c) { return 2; }
    public int f(double a, float b, long c) { return 3; }
    public int f(double a, int b, double c) { return 4; }
}

public class Test {
    public static void main(String[] args) {

        A alfa = new A();

        System.out.println(alfa.f(1, 2, 3));
        System.out.println(alfa.f(1.0, 2, 3));
        System.out.println(alfa.f(1, 2.0, 3));
        System.out.println(alfa.f(1.0, 2, 3.0));
        System.out.println(true || (1753/81 < 10235/473));
    }
}

```

- Indicare l'output del programma. Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione. (*13 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*12 punti*)

92. (2008-7-9)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a, double x);
    public String f(B b, float x) { return "A2"; }
}
class B extends A {
    public String f(A a, double x) { return "B1"; }
    public String f(C c, double x) { return "B2"; }
    private String f(Object o, double x) { return "B3"; }
}
class C extends B {
    public String f(A a, double x){ return "C1"; }
    public String f(B b, float x) { return "C2"; }
}

public class Test {

```

```

public static void main(String[] args) {
    C gamma = new C();
    B beta = gamma;
    A alfa = gamma;
    System.out.println(alfa.f(gamma, 5));
    System.out.println(beta.f(alfa, 7));
    System.out.println(gamma.f(gamma, 2.0));
    System.out.println(2 > 1);
}
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

93. (2008-6-19)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A a1, A a2);
    public String f(B bb, C cc) { return "A2"; }
}
class B extends A {
    public String f(A a1, A a2) { return "B1"; }
    public String f(C cc, B bb) { return "B2"; }
    private String f(Object x, B bb) { return "B3"; }
}
class C extends B {
    public String f(C c1, C c2) { return "C1"; }
    public String f(A a1, A a2) { return "C2"; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, alfa));
        System.out.println(alfa.f(beta, beta));
        System.out.println(beta.f(beta, gamma));
        System.out.println(gamma.f(gamma, gamma));
        System.out.println(7 & 3);
    }
}

```

- Indicare l'output del programma. (*15 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*12 punti*)

94. (2008-4-21)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(Object other, int n);
    public String f(B other, long n) { return "A2:" + n; }
}
class B extends A {
    public String f(Object other, int n) { return "B1:" + n; }
    private String f(C other, long n) { return "B2:" + n; }
}

```

```

class C extends B {
    public String f(Object other, long n) { return "C1:" + n; }
    public String f(C other, long n) { return "C2:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(alfa, 4));
        System.out.println(alfa.f(beta, 4L));
        System.out.println(beta.f(gamma, 4L));
        System.out.println(gamma.f(null, 3L));
        System.out.println(175 & 175);
    }
}

```

- Indicare l'output del programma. (*15 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*12 punti*)

95. (2008-3-27)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    private int f(double a, int b, A c) { return 1; }
    public int f(double a, float b, A c) { return 20; }
    public int f(long a, float b, B c) { return 10; }
}
class B extends A {
    public int f(double a, float b, A c) { return 30; }
    public int f(int a, int b, B c) { return 40; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa = beta;

        System.out.println(alfa.f(1, 2, alfa));
        System.out.println(alfa.f(1, 2, null));
        System.out.println(beta.f(1, 2, beta));
        System.out.println(beta.f(1.0, 2, beta));
        System.out.println(1234 & 1234);
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

96. (2008-2-25)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int a, double b) { return 1 + f(8, 8); }
    public int f(float a, double b) { return 1 + f(7, b); }
    private int f(int a, float b) { return 1; }
}
class B extends A {
}

```

```

        public int f(int a, double b) { return 4; }
        public int f(double a, double b) { return 5; }
    }
    public class Test {
        public static void main(String[] args) {
            B beta = new B();
            A alfa1 = new B();
            A alfa2 = new A();

            System.out.println(alfa1.f(1,2));
            System.out.println(alfa2.f(1,2));
            System.out.println(beta.f(1.0,2));
            System.out.println(8 | 1);
        }
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

97. (2008-1-30)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int a, float b, double c) { return 1; }
    public int f(double a, double b, double c) { return 2; }
    private int f(int a, double b, double c) { return 3; }
}
class B extends A {
    public int f(int a, float b, double c) { return 4; }
    public int f(int a, float b, int c) { return 5; }
}
public class Test {
    public static void main(String[] args) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println(alfa1.f(1,2,3));
        System.out.println(alfa2.f(1,2,3));
        System.out.println(beta.f(1.0,2,3));
        System.out.println(177 & 2);
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

98. (2007-9-17)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    private int f(int a, double b, double c) { return 1; }
    public int f(int a, float b, double c) { return 10; }
    public int f(double a, double b, double c) { return 20; }
}
class B extends A {
    public int f(int a, float b, int c) { return 15; }
    public int f(int a, float b, double c) { return 25; }
}

```

```

    }
    public class Test {
        public static void main( String [] args ) {
            B beta = new B();
            A alfa1 = beta;
            A alfa2 = new A();

            System.out.println( alfa1.f(1,2,3));
            System.out.println( alfa2.f(1,2,3));
            System.out.println( beta.f(1,2,3));
            System.out.println( beta.f(1.0,2,3));
            System.out.println( 7 / 2);
        }
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

99. (2007-7-20)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public static int x = 0;
    public A() { x++; }

    private int f(int a, double b) { return x; }
    public int f(int a, float b) { return x+5; }
    public int f(double a, double b) { return x+20; }
    public String toString() { return f(x, x) + ""; }
}

class B extends A {
    public int f(int a, float b) { return x-5; }
    public int f(int a, int b) { return x-10; }
}

public class Test {
    public static void main( String [] args ) {
        B beta = new B();
        A alfa1 = beta;
        A alfa2 = new A();

        System.out.println( alfa1 );
        System.out.println( alfa2 );
        System.out.println( beta );
        System.out.println( beta.f(4, 5.0) );
        System.out.println( 322 | 1 );
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

100. (2007-6-29)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

class A {
    public int f(int x, A a) { return 0; }
    public int f(double x, B b) { return 7; }
    public int f(double x, A a) { return 10; }
}

```

```

    }
    class B extends A {
        public int f(int x, B b) { return f(2.0 * x, b) + 1; }
        public int f(double x, B b) { return 20; }
        public int f(double x, A a) { return f((int) x, a) + 1; }
    }
    public class Test2 {
        public static void main(String[] args) {
            B beta = new B();
            A alfa = beta;

            System.out.println(alfa.f(3.0, beta));
            System.out.println(alfa.f(3.0, alfa));
            System.out.println(beta.f(3, beta));
            System.out.println(beta.f(3, alfa));
        }
    }
}

```

- Indicare l'output del programma.
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate.

101. (2007-4-26)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

abstract class A {
    public abstract String f(A other, int n);
    public String f(B other, long n) { return "A2:" + n; }
}
class B extends A {
    public String f(A other, int n) { return "B1:" + n; }
    private String f(C other, long n) { return "B2:" + n; }
}
class C extends B {
    public String f(A other, long n) { return "C1:" + n; }
    public String f(C other, long n) { return "C2:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(15 & 7);
        System.out.println(alfa.f(alfa, 4));
        System.out.println(alfa.f(beta, 4L));
        System.out.println(beta.f(gamma, 4L));
        System.out.println(gamma.f(gamma, 3L));
    }
}

```

- Indicare l'output del programma. (*20 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*5 punti*)

102. (2007-2-7)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public class A {
    public String f(A other, int n) { return "A:" + n; }
}

```

```

}
public class B extends A {
    public String f(Object other, int n) { return "B1:" + n; }
    public String f(B other, long n) { return "B2:" + n; }
}
public class C extends B {
    public String f(C other, boolean n) { return "C:" + n; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;

        System.out.println(alfa.f(beta, 3));
        System.out.println(beta.f(alfa, 3));
        System.out.println(gamma.f(gamma, 3));
        System.out.println(gamma.f(gamma, 3L));
    }
}

```

- Indicare l'output del programma. (*20 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*5 punti*)

103. (2007-2-23)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public abstract class A {
    public abstract String f(A other, int n);
    public String f(A other, long n) { return "A:" + n; }
}
public class B extends A {
    public String f(A other, int n) { return "B1:" + n; }
    public String f(Object other, int n) { return "B2:" + n; }
    public String f(B other, long n) { return "B3:" + n; }
}
public class C extends B {
    public String f(B other, long n) { return "C1:" + n; }
    public String f(C other, int n) { return "C2:" + n; }
}

public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = gamma;
        A alfa = gamma;
        System.out.println(alfa.f(beta, 4));
        System.out.println(alfa.f(beta, 4L));
        System.out.println(beta.f((Object) alfa, 4));
        System.out.println(gamma.f(gamma, 3));
    }
}

```

- Indicare l'output del programma. (*20 punti*)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (*5 punti*)

104. (2007-1-12)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public class A {
    public String f(A other, int n) { return "A:" + n; }
}
public class B extends A {
    public String f(A other, int n) { return "B1:" + n; }
    public String f(A other, long n) { return "B2:" + n; }
}
public class C extends B {
    public String f(Object other, int n) { return "C:" + n; }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.f(beta, 3));
        System.out.println(beta.f(alfa, 3));
        System.out.println(gamma.f(alfa, 3));
        System.out.println(alfa.equals(gamma));
    }
}

```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

105. (2006-9-15)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```

public class A {
    public boolean equals(A other) {
        System.out.println("in_A:");
        return true;
    }
}
public class B extends A {
    public boolean equals(A other) {
        System.out.println("in_B:");
        return true;
    }
}
public class C extends B {
    public boolean equals(Object other) {
        System.out.println("in_C:");
        return true;
    }
}
public class Test {
    public static void main(String[] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.equals(beta));
        System.out.println(beta.equals(alfa));
        System.out.println(gamma.equals(alfa));
        System.out.println(gamma.equals(new String("ciao")));
        System.out.println(15 & 1);
    }
}

```

- Indicare l'output del programma. (20 punti)

- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

106. (2006-7-17)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
public class A {
    public boolean equals(A other) {
        System.out.println("in_A:");
        return true;
    }
}
public class B extends A { }
public class C extends A {
    public boolean equals(Object other) {
        System.out.println("in_C:");
        return false;
    }
}
public class Test {
    public static void main(String [] args) {
        C gamma = new C();
        B beta = new B();
        A alfa = gamma;
        System.out.println(alfa.equals(beta));
        System.out.println(gamma.equals(beta));
        System.out.println(beta.equals(alfa));
        System.out.println(beta.equals((Object) alfa));
        System.out.println("true" + true);
    }
}
```

- Indicare l'output del programma. (20 punti)
- Per ogni chiamata ad un metodo (escluso `System.out.println`) indicare la lista delle firme candidate. (5 punti)

107. (2006-6-26)

Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
public class A {
    public int f() { return 1; }
    public int f(A x) { return f() + 1; }
}
public class B extends A {
    public int f() { return 3; }
    public int f(B x) { return f() + 10; }
}
public class C extends B {
    public int f(C x) { return 5; }
}
public class Test {
    public static void main(String [] args) {
        C gamma = new C();
        B beta = gamma;
        System.out.println(beta.f(beta));
        System.out.println(gamma.f(beta));
        System.out.println(523 < 523);
        System.out.println(257 & 1);
    }
}
```

- Indicare l'output del programma. (*15 punti*)
- Per ogni chiamata ad un metodo (escluso System.out.println) indicare la lista delle firme candidate. (*5 punti*)

108. (2006-4-27)

Dato il seguente programma:

```
public class A {  
    private int f() { return 1; }  
    public int f(int x) { return f() + 1; }  
}  
public class B extends A {  
    public int f(boolean x) { return 3; }  
    public int f(double x) { return f(true) + 1; }  
}  
public class C extends B {  
    public int f(boolean x) { return 5; }  
}  
public class Test {  
    public static void main(String[] args) {  
        B beta = new C();  
        System.out.println(beta.f(1));  
        System.out.println(beta.f(1.0));  
        System.out.println(523 & 523);  
        System.out.println(257 | 257);  
    }  
}
```

- Indicare l'output del programma. (*15 punti*)
- Per ogni chiamata ad un metodo (escluso System.out.println) indicare la lista delle firme candidate. (*5 punti*)

3 Esercizi elementari

109. (Angle, 2022-3-28)

Implementare la classe `Angle`, che rappresenta un angolo, e le sottoclassi `Acute`, `Right`, e `Obtuse`.

Gli angoli si misurano in gradi sessagesimali e devono essere dotati di ordinamento naturale. La classe `Angle` offre il metodo statico `of`, che crea un angolo del valore dato e del tipo corrispondente. Ad esempio, la chiamata `Angle.of(30)` deve restituire un oggetto di tipo effettivo `Acute`.

Le sottoclassi `Acute` e `Obtuse` hanno un costruttore pubblico che accetta il valore dell'angolo. La classe `Right`, invece, deve essere un Singleton, la cui unica istanza è accessibile solo tramite la chiamata `Angle.of(90)`. Infine, non deve essere possibile istanziare oggetti di tipo effettivo `Angle`.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Angle a = Angle.of(30.0);
Angle.Acute b = new Angle.Acute(20.0);
Angle c = Angle.of(90.0);
Angle d = Angle.of(90.0);
Angle.Obtuse e = (Angle.Obtuse) Angle.of(120.5);
System.out.println(a == b);
System.out.println(a.getClass() == b.getClass());
System.out.println(c == d);
System.out.println(a.compareTo(b));
```

Output:

```
false
true
true
1
```

110. (FilteredSet, 2022-2-24)

Si ricordi l'interfaccia standard `Predicate`:

```
public interface Predicate<T> {
    boolean test(T t);
}
```

Realizzare la classe `FilteredSet` che rappresenta un set che può contenere solo oggetti sui quali un dato predicato restituisce `true`. Il costruttore di `FilteredSet` accetta il predicato che farà da filtro. La classe deve offrire i metodi `add`, `contains` e `remove` tipici dell'interfaccia `Collection`, assicurandosi che `add` fallisca se l'oggetto che si tenta di inserire non soddisfa il predicato.

Infine, il metodo `intersect` accetta un altro `FilteredSet` "other" e restituisce un nuovo `FilteredSet` che contiene gli elementi comuni a `this` e `other`. Il filtro del nuovo `FilteredSet` deve essere l'*and logico* dei due filtri di `this` e `other`. Ovvero, il nuovo `FilteredSet` deve accettare solo oggetti che verrebbero accettati sia da `this` che da `other`.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
FilteredSet<String> s1 = new FilteredSet<>(x -> x.length() > 2);
FilteredSet<String> s2 = new FilteredSet<>(x -> x.indexOf("a") != -1);
System.out.println(s1.add("ciao"));
System.out.println(s1.add("aa"));
System.out.println(s2.add("ciao"));
FilteredSet<String> s3 = s1.intersect(s2);
System.out.println(s3.add("pippo"));
```

Output:

```
true
false
true
false
```

111. (Exchange, 2022-1-26)

Realizzare la classe Exchange che rappresenta una borsa valori. Il metodo setPrice imposta il prezzo corrente di un titolo quotato. Il metodo addLowAlert fa in modo che un dato runnable venga eseguito la prima volta che il prezzo di un dato titolo raggiunge o scende sotto una data soglia. Il runnable viene eseguito immediatamente se il prezzo di quel titolo è già inferiore o uguale alla soglia. Simmetricamente, il metodo addHighAlert offre lo stesso servizio, quando il prezzo raggiunge o sale al di sopra di una data soglia. E' possibile impostare più alert per lo stesso titolo, con soglie diverse o uguali tra loro.

La classe Exchange deve essere *thread-safe*.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Exchange borsa = new Exchange();
borsa.setPrice("MaxiCom", 10.56);
borsa.setPrice("MegaCorp", 18.2);
borsa.setPrice("SuperMarkt", 3.91);
borsa.addLowAlert("MegaCorp", 17.5, () -> { System.out.println("Below_the_threshold!"); });
borsa.addHighAlert("MaxiCom", 12, () -> { System.out.println("More_than_12!"); });
borsa.addHighAlert("MaxiCom", 20.5, () -> { System.out.println("More_than_20.5!"); });
borsa.setPrice("MaxiCom", 12.3);
```

Output:

```
More than 12!
```

112. (Radio, 2021-9-24)

Realizzare le classi Radio e Channel che rappresentano una radio e una stazione radiofonica. La classe Radio offre un costruttore senza argomenti e i seguenti metodi:

- addChannel memorizza e restituisce una nuova stazione, caratterizzata da nome e frequenza. Il tentativo di memorizzare una stazione che ha la stessa frequenza di una stazione già memorizzata deve provocare un'eccezione.
- nearest accetta una frequenza e restituisce la stazione con la frequenza più vicina a quella data.

Inoltre, se si itera su un oggetto Radio si ottiene la sequenza di stazioni inserite, *in ordine crescente di frequenza*.

Fare in modo che l'unico modo per creare oggetti Channel sia tramite il metodo addChannel.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|---|---|
| <p>Esempio d'uso:</p> <pre>Radio r = new Radio(); Radio.Channel rai1 = r.addChannel("Rai_Radio_Uno", 89.3); Radio.Channel kk = r.addChannel("Radio_Kiss_Kiss", 101.4); Radio.Channel rmc = r.addChannel("Radio_Monte_Carlo", 96.4) ; for (Radio.Channel c: r) { System.out.println(c); } System.out.println(r.nearest(98.1));</pre> | <p>Output:</p> <p>Rai Radio Uno (89.3) Radio Monte Carlo (96.4) Radio Kiss Kiss (101.4)</p> <p>Radio Monte Carlo (96.4)</p> |
|---|---|

113. (GreenPass, 2021-7-26)

Realizzare le classi Person e GreenPass che rappresentano una persona e una certificazione verde. Una persona è identificata dal suo nome. Il metodo `vaccinate` di Person accetta come argomento la data di vaccinazione (un intero che rappresenta un giorno) e restituisce un oggetto GreenPass. La classe GreenPass offre i seguenti metodi:

- `isValidOn` accetta una data e restituisce vero se questa certificazione verde è valida in quella data.
- `belongsTo` accetta un Person e restituisce vero se questa certificazione appartiene a quella persona.

La validità di un GreenPass è definita dalle seguenti regole:

- se si tratta della prima dose (prima chiamata a `vaccinate` per questa persona), il GreenPass è valido per 180 giorni;
- negli altri casi, il GreenPass è valido per 270 giorni.

Opzionale: Fare in modo che l'unico modo per creare oggetti GreenPass sia tramite il metodo `vaccinate` (qualsiasi altro tentativo deve provocare errore di compilazione o eccezione a runtime).

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre>Person aldo = new Person("Aldo"), barbara = new Person("Barbara"); GreenPass p1 = aldo.vaccinate(10), p2 = aldo.vaccinate(250); System.out.println(p1.isValidOn(20)); System.out.println(p1.isValidOn(200)); System.out.println(p1.belongsTo(barbara));</pre> | <p>Output:</p> <p>true false false</p> |
|--|--|

114. (WiFi, 2021-10-26)

Realizzare le classi WiFi e Network, che rappresentano un elenco di reti WiFi e una singola rete. La classe WiFi offre un costruttore senza argomenti e i seguenti metodi:

- `addNetwork`: memorizza e restituisce una nuova rete, caratterizzata da nome (SSID) e intensità del segnale.¹
- `strongest`: restituisce la rete con l'intensità più alta (più vicina allo zero).

Inoltre, gli oggetti WiFi devono essere iterabili, dando la possibilità di scorrere le reti inserite, *in ordine di intensità decrescente*.

La classe Network offre soltanto il metodo `updateStrength`, che aggiorna l'intensità del segnale.

Fare in modo che l'unico modo per creare oggetti Network sia tramite il metodo `addNetwork`.

L'implementazione deve rispettare il seguente esempio d'uso.

¹Misurata in dBm (decibel-milliwatt), l'intensità è un numero negativo ed il segnale è più intenso quanto più il valore è vicino allo zero.

| | |
|--|---------------------------|
| Esempio d'uso: | Output: |
| <pre>WiFi manager = new WiFi(); WiFi.Network home = manager.addNetwork("Vodafone", -40.5); WiFi.Network hotel = manager.addNetwork("Hotel_Vesuvio", -53.05); WiFi.Network neighbor = manager.addNetwork("Casa_Esposito", -48.95); neighbor.updateStrength(-39.6); WiFi.Network x = manager.strongest(); System.out.println(x);</pre> | Casa Esposito (-39.6 dBm) |

115. (Cartella, 2017-7-20)

Realizzare la classe **Cartella**, che rappresenta una cartella nella Tombola. Una cartella contiene 15 numeri casuali diversi, compresi tra 1 e 90, disposti in 3 righe di 5 numeri, rispettando la seguente regola:

- una riga non può contenere due numeri della stessa “decina”; ad esempio, una riga può contenere 9 e 11, ma non 11 e 13.

Il metodo **segna** accetta il prossimo numero estratto, e controlla se questa cartella ha ottenuto un premio, restituendo **null**, oppure un valore enumerato che rappresenta uno dei premi della Tombola: AMBO, TERNO, QUATERNA, CINQUINA, TOMBOLA (implementare anche questa enumerazione).

L’implementazione deve rispettare il seguente esempio d’uso:

| | |
|--|---|
| Esempio d'uso: | Un output possibile: |
| <pre>Cartella c = new Cartella(); System.out.println(c.segna(1)); System.out.println(c.segna(2)); System.out.println(c.segna(12)); System.out.println(c.segna(22)); System.out.println(c.segna(82));</pre> | null null null AMBO null |

116. (Book, 2016-7-21)

Implementare la classe Book, che rappresenta un libro diviso in capitoli. Il metodo addChapter aggiunge un capitolo in coda al libro, caratterizzato da titolo e contenuto. I capitoli sono automaticamente numerati a partire da 1. Il metodo getChapterName(i) restituisce il titolo del capitolo *i*-esimo, mentre il metodo getChapterContent(i) ne restituisce il contenuto.

Gli oggetti Book devono essere clonabili. Inoltre, la classe deve essere dotata di ordinamento naturale, basato sul numero di capitoli.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Book b = new Book();
b.addChapter("Prefazione", "Sono_passati_pochi_anni...");
b.addChapter("Introduzione", "Un_calcolatore_digitale...")
;
b.addChapter("Sistemi_di_elaborazione", "Un_calcolatore...
");
Book bb = b.clone();
System.out.println(bb.getChapterContent(1));
System.out.println(bb.getChapterTitle(2));
```

Output:

Sono passati pochi anni...
Introduzione

117. (GameLevel, 2016-3-3)

Implementare la classe GameLevel, che rappresenta un livello in un gioco 2D, in cui un personaggio si muove su una griglia di caselle. Il costruttore accetta le dimensioni del livello (larghezza e altezza). Il metodo setWall accetta le coordinate di una casella e mette un muro in quella casella. Il metodo areConnected accetta le coordinate di due caselle e restituisce *vero* se e solo se esiste un percorso tra di loro.

Caso d'uso:

```
GameLevel map = new GameLevel(3, 3);
System.out.println(map.areConnected(0,0,2,2));
map.setWall(0,1);
map.setWall(1,1);
System.out.println(map.areConnected(0,0,2,2));
map.setWall(2,1);
System.out.println(map.areConnected(0,0,2,2));
```

Output:

true true false

118. (Question e Answer, 2015-7-8)

Per un sito di domande e risposte, realizzare le classi Question e Answer. Ogni risposta è associata ad un'unica domanda e gli utenti possono votare la risposta migliore invocando il metodo voteUp di Answer. Inoltre, il metodo getBestAnswer restituisce *in tempo costante* la risposta (o una delle risposte) che ha ottenuto il maggior numero di voti.

Rispettare il seguente caso d'uso.

Caso d'uso:

```
Question q = new Question("Dove_si_trova_Albuquerque?");
Answer a1 = new Answer(q, "Canada");
Answer a2 = new Answer(q, "New_Mexico");
a1.voteUp();
System.out.println(q.getBestAnswer());
a2.voteUp();
a2.voteUp();
System.out.println(q.getBestAnswer());
```

Output:

```
Canada
New Mexico
```

119. (Box, 2015-2-5)

Realizzare la classe **Box**, che rappresenta una scatola, caratterizzata dalle sue tre dimensioni: altezza, larghezza e profondità. Due scatole sono considerate uguali (da `equals`) se hanno le stesse dimensioni. Le scatole sono dotate di ordinamento naturale basato sul loro volume. Infine, il metodo `fitsIn`, invocato su una scatola *x*, accetta un'altra scatola *y* e restituisce `true` se e solo se *y* è sufficientemente grande da contenere *x*.

Esempio d'uso:

```
Box grande = new Box(20, 30, 40), grande2 = new Box(30, 20, 40),
piccolo = new Box(10, 10, 50);
System.out.println(grande.equals(grande2));
System.out.println(grande.compareTo(piccolo));
System.out.println(piccolo . fitsIn (grande));
```

Output:

```
false
1
false
```

120. (Playlist, 2014-7-28)

Implementare le classi **Song** e **Playlist**. Una canzone è caratterizzata dal nome e dalla durata in secondi. Una playlist è una lista di canzoni, compresi eventuali duplicati, ed offre il metodo `add`, che aggiunge una canzone in coda alla lista, e `remove`, che rimuove *tutte* le occorrenze di una canzone dalla lista. Infine, la classe **Playlist** è dotata di ordinamento naturale basato sulla durata totale di ciascuna playlist.

Sono preferibili le implementazioni in cui il confronto tra due playlist avvenga in tempo costante.

Esempio d'uso:

```
Song one = new Song("One", 275), two = new Song("Two", 362);
Playlist a = new Playlist(), b = new Playlist();
a.add(one); a.add(two); a.add(one);
b.add(one); b.add(two);
System.out.println(a.compareTo(b));
a.remove(one);
System.out.println(a.compareTo(b));
```

Output:

```
1
-1
```

121. (Safe, 2012-4-23)

Implementare la classe **Safe**, che rappresenta una cassaforte che contiene una stringa segreta, protetta da un numero intero che funge da combinazione. Il costruttore accetta la combinazione e la stringa segreta. Il metodo `open` accetta un numero intero e restituisce la stringa segreta se tale numero coincide con la combinazione. Altrimenti, restituisce `null`. Infine, se le ultime 3 chiamate a `open` sono fallite, la cassaforte diventa irreversibilmente **bloccata** ed ogni ulteriore operazione solleva un'eccezione.

Implementare la classe **ResettableSafe** come una sottoclasse di **Safe** che aggiunge il metodo `changeKey`, che accetta due interi *old* e *new* e restituisce un boolean. Se la cassaforte è bloccata, il metodo solleva un'eccezione. Altrimenti, se l'argomento *old* coincide con la combinazione attuale, il metodo imposta la combinazione della cassaforte a *new* e restituisce true. Se invece *old* differisce dalla combinazione attuale, il metodo restituisce false.

Una `ResettableSafe` diventa bloccata dopo tre tentativi falliti di `open` o di `changeKey`. Ogni chiamata corretta a `open` o `changeKey` azzera il conteggio dei tentativi falliti.

Suggerimento: prestare attenzione alla scelta della visibilità di campi e metodi.

| | |
|---|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>ResettableSafe s = new ResettableSafe(2381313, "L'assassino_e", _il_maggiordomo."); System.out.println(s.open(887313)); System.out.println(s.open(13012)); System.out.println(s.changeKey(12,34)); System.out.println(s.open(2381313));</pre> | <pre>null null false Exception in thread "main"...</pre> |

122. (PrintBytes, 2011-3-4)

Scrivere un metodo statico `printBytes`, che prende come argomento un `long` che rappresenta un numero di byte minore di 10^{15} , e restituisce una stringa in cui il numero di byte viene riportato nell'unità di misura più comoda per la lettura, tra: bytes, kB, MB, GB, e TB. Più precisamente, il metodo deve individuare l'unità che permetta di esprimere (approssimativamente) il numero di byte dato utilizzando tre cifre intere e una frazionaria. L'approssimazione può essere per troncamento oppure per arrotondamento.

| input | output |
|-------------|-------------|
| 123 | "123 bytes" |
| 3000 | "3.0 kB" |
| 19199 | "19.1 kB" |
| 12500000 | "12.5 MB" |
| 710280000 | "710.2 MB" |
| 72000538000 | "72.0 GB" |

123. (Time, 2010-9-14)

Implementare la classe `Time`, che rappresenta un orario della giornata (dalle 00:00:00 alle 23:59:59). Gli orari devono essere confrontabili secondo `Comparable`. Il metodo `minus` accetta un altro orario `x` come argomento e restituisce la differenza tra questo orario e `x`, sotto forma di un nuovo oggetto `Time`. La classe fornisce anche gli orari predefiniti `MIDDAY` e `MIDNIGHT`.

| | |
|--|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Time t1 = new Time(14,35,0); Time t2 = new Time(7,10,30); Time t3 = t1.minus(t2); System.out.println(t3); System.out.println(t3.compareTo(t2)); System.out.println(t3.compareTo(Time.MIDDAY));</pre> | <pre>7:24:30 1 -1</pre> |

124. (Tetris, 2010-7-26)

Il Tetris è un videogioco il cui scopo è incastrare tra loro pezzi bidimensionali di 7 forme predefinite, all'interno di uno schema rettangolare. Implementare la classe astratta `Piece`, che rappresenta un generico pezzo, e le sottoclassi concrete `T` ed `L`, che rappresentano i pezzi dalla forma omonima.

La classe `Piece` deve offrire i metodi `put`, che aggiunge questo pezzo alle coordinate date di un dato schema, e il metodo `rotate`, che ruota il pezzo di 90 gradi in senso orario (senza modificare alcuno schema). Il metodo `put` deve lanciare un'eccezione se non c'è posto per questo pezzo alle coordinate date. Uno schema viene rappresentato da un array bidimensionale di valori booleani (`false` per libero, `true` per occupato).

E' opportuno raccogliere quante più funzionalità è possibile all'interno della classe `Piece`. Il seguente caso d'uso assume che `print_board` sia un opportuno metodo per stampare uno schema.

| | |
|--|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>boolean board[][] = new boolean[5][12]; Piece p1 = new T(); p1.put(board, 0, 0); Piece p2 = new L(); p2.put(board, 0, 4); print_board(board); p2.rotate(); p2.put(board, 2, 7); print_board(board);</pre> | <pre>----- X X XXX X XX ----- X X XXX X XX XXX X</pre> |

125. (Crosswords, 2010-5-3)

Si implementi la classe `Crosswords`, che rappresenta uno schema di parole crociate, inizialmente vuoto. Il costruttore accetta le dimensioni dello schema. Il metodo `addWord` aggiunge una parola allo schema e restituisce `true`, a patto che la parola sia *compatibile* con quelle precedentemente inserite; altrimenti, restituisce `false` senza modificare lo schema. Il metodo prende come argomenti le coordinate iniziali della parola, la parola stessa e la direzione (H per orizzontale e V per verticale).

Le regole di *compatibilità* sono:

- Una parola non si può sovrapporre ad un'altra della stessa direzione.
- Una parola si può incrociare con un'altra solo su di una lettera comune.
- Ogni parola deve essere preceduta e seguita da un bordo o da una casella vuota.

Non è necessario implementare il metodo `toString`. E' opportuno implementare le direzioni H e V in modo che siano le uniche istanze del loro tipo.

Suggerimenti:

- Per evitare di scrivere separatamente i due casi per orizzontale e verticale, è possibile aggiungere i metodi `getChar/setChar`, che prendono come argomenti una riga r , una colonna c , una direzione d (H o V) e un offset x , e leggono o scrivono il carattere situato a distanza x dalla casella r, c , in direzione d .
- Il metodo `s.charAt(i)` restituisce il carattere i -esimo della stringa `s` (per i compreso tra 0 e `s.length()-1`).

| | |
|--|---|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Crosswords c = new Crosswords(6, 8); System.out.println(c.addWord(0,3, "casa", Crosswords.V)); System.out.println(c.addWord(2,1, "naso", Crosswords.H)); System.out.println(c.addWord(2,0, "pippo", Crosswords.H)); System.out.println(c);</pre> | <pre>true true false c a *naso* a *</pre> |

126. (Wall, 2010-2-24)

La classe `Wall` rappresenta un muro di mattoni, ciascuno lungo 10cm, poggiati l'uno sull'altro. Il costruttore accetta l'altezza massima (in file di mattoni) e la larghezza massima (in cm) del muro. Il metodo `addBrick` aggiunge un mattone alla fila e alla posizione (in cm) specificata, restituendo un oggetto di tipo `Brick`. Il metodo `isStable` della classe `Brick` restituisce vero se in quel momento questo mattone è in una posizione stabile, indipendentemente dai mattoni eventualmente poggiati *sopra* di esso.

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre>Wall w = new Wall(10, 100); w.addBrick(0,10); w.addBrick(0,30); Wall.Brick b3 = w.addBrick(1,2); Wall.Brick b4 = w.addBrick(1,13); Wall.Brick b5 = w.addBrick(1,36); System.out.println(b3.isStable()); System.out.println(b4.isStable()); System.out.println(b5.isStable()); w.addBrick(0,45); System.out.println(b5.isStable());</pre> | <p>Output dell'esempio d'uso: (Nota: l'esempio è accompagnato da una figura alla lavagna)</p> <pre>false true false true</pre> |
|--|--|

127. (Segment, 2010-11-30)

Implementare la classe `Segment`, che rappresenta un segmento collocato nel piano cartesiano. Il costruttore accetta le coordinate dei due vertici, nell'ordine x_1, y_1, x_2, y_2 . Il metodo `getDistance` restituisce la distanza tra la retta che contiene il segmento e l'origine del piano. Ridefinire il metodo `equals` in modo che due segmenti siano considerati uguali se hanno gli stessi vertici. Fare in modo che i segmenti siano clonabili.

Si ricordi che:

- L'area del triangolo con vertici di coordinate (x_1, y_1) , (x_2, y_2) e (x_3, y_3) è data da:

$$\frac{|x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)|}{2}.$$

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre>Segment s1 = new Segment(0.0, -3.0, 4.0, 0.0); Segment s2 = new Segment(4.0, 0.0, 0.0, -3.0); Segment s3 = s2.clone(); System.out.println(s1.equals(s2)); System.out.println(s1.getDistance());</pre> | <p>Output dell'esempio d'uso:</p> <pre>true 2.4</pre> |
|--|---|

128. (Color, 2010-1-22)

La classe `Color` rappresenta un colore, determinato dalle sue componenti RGB. La classe offre alcuni colori predefiniti, tra cui `RED`, `GREEN` e `BLUE`. Un colore nuovo si può creare solo con il metodo factory `make`. Se il client cerca di ricreare un colore predefinito, gli viene restituito quello e non uno nuovo. Ridefinire anche il metodo `toString`, in modo che rispetti il seguente caso d'uso.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre>Color rosso = Color.RED; Color giallo = Color.make(255, 255, 0); Color verde = Color.make(0, 255, 0); System.out.println(rosso); System.out.println(giallo); System.out.println(verde); System.out.println(verde == Color. GREEN);</pre> | <p>Output dell'esempio d'uso:</p> <pre>red (255, 255, 0) green true</pre> |
|--|---|

129. (Circle, 2009-4-23)

Nell'ambito di un programma di geometria, la classe `Circle` rappresenta una circonferenza sul piano cartesiano. Il suo costruttore accetta le coordinate del centro ed il valore del raggio. Il

metodo `overlaps` prende come argomento un'altra circonferenza e restituisce vero se e solo se le due circonferenze hanno almeno un punto in comune.

Fare in modo che `Circle` implementi `Comparable`, con il seguente criterio di ordinamento: una circonferenza è “minore” di un’altra se è interamente contenuta in essa, mentre se nessuna delle due circonferenze è contenuta nell’altra, esse sono considerate “uguali”. Dire se tale criterio di ordinamento è valido, giustificando la risposta.

| | |
|--|----------------------------|
| Esempio d’uso: | Output dell’esempio d’uso: |
| <pre>Circle c1 = new Circle(0,0,2); Circle c2 = new Circle(1,1,1); System.out.println(c1.overlaps(c2)); System.out.println(c1.compareTo(c2));</pre> | <pre>true 0</pre> |

130. (Interval, 2009-1-29)

Si implementi la classe `Interval`, che rappresenta un intervallo di numeri reali. Un intervallo può essere chiuso oppure aperto, sia a sinistra che a destra. Il metodo `contains` prende come argomento un numero x e restituisce vero se e solo se x appartiene a questo intervallo. Il metodo `join` restituisce l’unione di due intervalli, senza modificarli, sollevando un’eccezione nel caso in cui questa unione non sia un intervallo. Si implementino anche le classi `Open` e `Closed`, in modo da rispettare il seguente caso d’uso.

| | |
|--|--|
| Esempio d’uso: | Output dell’esempio d’uso: |
| <pre>Interval i1 = new Interval.Open(5, 10.5); Interval i2 = new Interval.Closed(7, 20); Interval i3 = i1.join(i2); System.out.println(i1.contains(5)); System.out.println(i1); System.out.println(i2); System.out.println(i3);</pre> | <pre>false (5, 10.5) [7, 20] (5, 20)</pre> |

131. (Anagramma, 2009-1-15)

Si implementi un metodo statico che prende come argomenti due stringhe e restituisce *vero* se sono l’una l’anagramma dell’altra e *falso* altrimenti.

132. (2008-9-8)

Discutere della differenza tra classi astratte ed interfacce. In particolare, illustrare le differenze relativamente ai costruttori, ai campi e ai metodi che possono (o non possono) contenere. Infine, illustrare le linee guida per la scelta dell’una o dell’altra tipologia. (Una pagina al massimo)

133. (Triangolo, 2008-4-21)

Nell’ambito di un programma di geometria, si implementi la classe `Triangolo`, il cui costruttore accetta le misure dei tre lati. Se tali misure non danno luogo ad un triangolo, il costruttore deve lanciare un’eccezione. Il metodo `getArea` restituisce l’area di questo triangolo. Si implementino anche la classe `Triangolo.Rettangolo`, il cui costruttore accetta le misure dei due cateti, e la classe `Triangolo.Iisoscele`, il cui costruttore accetta le misure della base e di uno degli altri lati.

Si ricordi che:

- Tre numeri a , b e c possono essere i lati di un triangolo a patto che $a < b + c$, $b < a + c$ e $c < a + b$.
- L’area di un triangolo di lati a , b e c è data da:

$$\sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)} \quad (\text{formula di Erone})$$

dove p è il semiperimetro.

| | |
|---|--|
| Esempio d'uso (fuori dalla classe Triangolo): <pre>Triangolo x = new Triangolo(10,20,25); Triangolo y = new Triangolo.Rettangolo(5,8); Triangolo z = new Triangolo.Iisoscele(6,5); System.out.println(x.getArea()); System.out.println(y.getArea()); System.out.println(z.getArea());</pre> | Output dell'esempio d'uso: 94.9918 19.9999 12.0 |
|---|--|

134. (Impianto e Apparecchio, 2008-3-27)

Si implementi una classe **Impianto** che rappresenta un impianto elettrico, e una classe **Apparecchio** che rappresenta un apparecchio elettrico collegabile ad un impianto. Un impianto è caratterizzato dalla sua potenza massima erogata (in Watt). Ciascun apparecchio è caratterizzato dalla sua potenza assorbita (in Watt). Per quanto riguarda la classe **Impianto**, il metodo **collega** collega un apparecchio a questo impianto, mentre il metodo **potenza** restituisce la potenza attualmente assorbita da tutti gli apparecchi *collegati* all'impianto ed *accesi*.

I metodi **on** e **off** di ciascun apparecchio accendono e spengono, rispettivamente, questo apparecchio. Se, accendendo un apparecchio col metodo **on**, viene superata la potenza dell'impianto a cui è collegato, deve essere lanciata una eccezione.

| | |
|---|---|
| Esempio d'uso: <pre>Apparecchio tv = new Apparecchio(150); Apparecchio radio = new Apparecchio(30); Impianto i = new Impianto(3000); i.collega(tv); i.collega(radio); System.out.println(i.potenza()); tv.on(); System.out.println(i.potenza()); radio.on(); System.out.println(i.potenza());</pre> | Output dell'esempio d'uso: 0 150 180 |
|---|---|

135. (2008-2-25)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private B myb;

    private B f(B b) {
        myb = new B(true + "true");
        int x = b.confronta(myb);
        int y = myb.confronta(b);
        return myb.valore();
    }

    private Object zzz = B.z;
}
```

136. (2008-1-30)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private B myb;
```

```

private int f(B b) {
    A x = B.copia(b);
    myb = B.copia(77);
    double d = myb.g();
    return myb.g();
}

private int x = B.x;
}

```

137. (Aereo, 2007-9-17)

Si implementi una classe `Aereo`. Ogni aereo si può trovare in ogni istante di tempo in uno dei seguenti quattro stati: in fase di decollo, in fase di crociera, in fase di atterraggio, atterrato. I quattro metodi `decollo`, `crociera`, `atterraggio`, `atterrato` cambiano lo stato dell'aereo. Questi metodi devono sollevare un'eccezione nuova, definita da voi, se non vengono chiamati nell'ordine giusto. Infine, il metodo `nvoli` restituisce il numero di voli completati dall'aereo fino a quel momento.

138. (ParkingLot, 2007-7-20)

Implementare una classe `ParkingLot`, che rappresenta un parcheggio con posti auto disposti secondo una griglia $m \times n$. Il costruttore prende come argomenti le dimensioni m ed n del parcheggio. Il metodo `carIn` aggiunge un veicolo al parcheggio e restituisce la riga e la colonna del posto assegnato al nuovo veicolo, oppure `null` se il parcheggio è pieno. Il metodo `carOut` prende come argomenti le coordinate di un veicolo che sta lasciando il parcheggio e restituisce il numero di secondi trascorsi dal veicolo nel parcheggio, oppure `null` se alle coordinate indicate non si trova alcun veicolo.

Suggerimento: utilizzare la classe `java.util.Date` per misurare il tempo.

| | |
|--|---|
| Esempio d'uso: <pre> ParkingLot p = new ParkingLot(10, 10); Pair<Integer> pos1 = p.carIn(); Pair<Integer> pos2 = p.carIn(); Thread.sleep(1000); int sec1 = p.carOut(pos1); Thread.sleep(1000); int sec2 = p.carOut(pos2); System.out.println("(" + pos1.getFirst() + "," + pos1.getSecond() + "), " + sec1); System.out.println("(" + pos2.getFirst() + "," + pos2.getSecond() + "), " + sec2); </pre> | Output: <code>(0, 0), 1</code> <code>(0, 1), 2</code> |
|--|---|

139. (Rational, 2007-6-29)

- (18 punti) Si implementi una classe `Rational` che rappresenti un numero razionale in maniera esatta. Il costruttore accetta numeratore e denominatore. Se il denominatore è negativo, viene lanciata una eccezione. Il metodo `plus` prende un altro `Rational` x come argomento e restituisce la somma di `this` e x . Il metodo `times` prende un altro `Rational` x come argomento e restituisce il prodotto di `this` e x .
- (9 punti) La classe deve assicurarsi che numeratore e denominatore siano sempre ridotti ai minimi termini. (Suggerimento: la minimizzazione della frazione può essere compito del costruttore)
- (7 punti) La classe deve implementare l'interfaccia `Comparable<Rational>`, in base al normale ordinamento tra razionali.

| | |
|---|---|
| Esempio d'uso: | Output dell'esempio d'u- so: 1/6 13/30 2/45 |
| <pre>Rational n = new Rational(2,12); // due dodicesimi Rational m = new Rational(4,15); // quattro quindicesimi Rational o = n.plus(m); Rational p = n.times(m); System.out.println(n); System.out.println(o); System.out.println(p);</pre> | |

140. (Genealogia, 2007-4-26)

Nell'ambito di un programma di genealogia, si implementi la classe (o interfaccia) Person e le sottoclassi Man e Woman, con le seguenti caratteristiche. Una persona è dotata di nome e cognome. Il metodo addChild di Person prende una persona x come argomento e segnala che x è figlia di this. Il metodo marries di Person prende una persona x come argomento e segnala che x è sposata con this. Il metodo marries lancia un'eccezione se x è dello stesso genere di this. Il metodo statico areSiblings prende come argomenti due persone x e y e restituisce vero se x ed y sono fratelli o sorelle e falso altrimenti.

| | |
|---|--|
| Esempio d'uso: | Output dell'esempio d'u- so: false true |
| <pre>Person a = new Man("Mario", "Rossi"); Person b = new Woman("Luisa", "Verdi"); Person c = new Man("Luca", "Rossi"); Person d = new Woman("Anna", "Rossi"); Person e = new Woman("Daniela", "Rossi"); a.marries(b); a.addChild(c); b.addChild(d); c.addChild(e); System.out.println(Person.areSiblings(a, b)); System.out.println(Person.areSiblings(c, d));</pre> | |

141. (2007-4-26)

Individuare gli errori di compilazione nel seguente programma. Commentare brevemente ciascun errore e fornire una possibile correzione.

```

1 public class Errors {
2     private static int sval = 7;
3     private int val = sval;
4
5     public Errors() { super(); }
6
7     private class A {
8         private A(int n) { val += n; }
9     }
10    private class B extends A {
11        B() { val = sval; }
12    }
13
14    public static void main(String[] args) {
15        Errors t = new Errors();
16        A a = t.new A(5);
17        B b = a.new B();
18    }
19 }
```

142. (Polinomio, 2007-1-12)

Un polinomio è una espressione algebrica del tipo $a_0 + a_1x + \dots + a_nx^n$. Si implementi una classe **Polynomial**, dotata di un costruttore che accetta un array contenente i coefficienti $a_0 \dots a_n$, e dei seguenti metodi: **getDegree** restituisce il grado del polinomio; **times** accetta un polinomio **p** come argomento e restituisce un polinomio che rappresenta il prodotto di **this** e **p**; **toString** produce una stringa simile a quella mostrata nel seguente caso d'uso.

| | |
|--|---|
| Esempio d'uso: <pre>double a1[] = {1, 2, 3}; double a2[] = {2, 2}; Polynomial p1 = new Polynomial(a1); Polynomial p2 = new Polynomial(a2); Polynomial p3 = p1.times(p2); System.out.println(p1); System.out.println(p2); System.out.println(p3);</pre> | Output dell'esempio d'uso: <pre>1.0 + 2.0 x^1 + 3.0 x^2 2.0 + 2.0 x^1 2.0 + 6.0 x^1 + 10.0 x^2 + 6.0 x^3</pre> |
|--|---|

143. (FallingBody, 2006-9-15)

Nel contesto di un programma di simulazione per la cinematica, si implementi una classe **FallingBody** che rappresenta un corpo puntiforme dotato di massa, che cade soggetto solo alla forza di gravità terrestre. Il costruttore della classe prende come argomento la massa del corpo e la sua altezza iniziale. Si supponga che tutte le grandezze siano espresse in unità tra loro omogenee (altezza in metri, velocità in metri al secondo, etc.). Il metodo **progress** simula il passaggio di un dato numero di secondi. Il metodo **toString** va ridefinito in modo da mostrare l'altezza dal suolo e la velocità corrente del corpo. Non deve essere possibile creare sottoclassi di **FallingBody**.

Si supponga che l'accelerazione di gravità sia pari a $10 \frac{m}{s^2}$. Si ricordano le equazioni del moto uniformemente accelerato.

$$v = v_0 + at;$$

$$s = s_0 + v_0t + \frac{1}{2}at^2.$$

| | |
|--|---|
| Esempio d'uso: <pre>// Corpo di 2 kili , ad un'altezza di 20 metri. FallingBody b = new FallingBody(2, 20) ; System.out.println(b); b.progress(1); System.out.println(b); b.progress(1); System.out.println(b); b.progress(7); System.out.println(b);</pre> | Output dell'esempio d'uso: <pre>altezza: 20.0, velocita': 0.0 altezza: 15.0, velocita': 10.0 altezza: 0.0, velocita': 0.0 altezza: 0.0, velocita': 0.0</pre> |
|--|---|

144. (TreeType, 2006-9-15)

Implementare le classi **TreeType** e **Tree**. **TreeType** rappresenta un tipo di albero (pino, melo, etc.), mentre **Tree** rappresenta un particolare esemplare di albero. Ogni **TreeType** è caratterizzato dal suo nome. Ogni **Tree** ha un tipo base ed eventualmente degli innesti di altri tipi di alberi. Il metodo **addGraft** di **Tree** aggiunge un innesto ad un albero, purchè non sia dello stesso tipo dell'albero stesso. Il metodo **getCounter** di **Tree** restituisce il numero di alberi che sono stati creati. Il metodo **getCounter** di **TreeType** restituisce il numero di alberi di quel tipo che sono stati creati. (32 punti)

Ridefinire il metodo **clone** di **Tree**, facendo attenzione ad eseguire una copia profonda laddove sia necessario. (8 punti)

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre> TreeType melo = new TreeType("melo"); TreeType pero = new TreeType("pero" ; Tree unMelo = new Tree(melo); Tree unAltroMelo = new Tree(melo); unAltroMelo.addGraft(pero); unAltroMelo.addGraft(pero); System.out.println("Sono_stati_creati_" + melo.getCounter() + "_meli_fino_" + "a_questo_momento."); System.out.println("Sono_stati_creati_" + Tree.getCounter() + "_alberi_fino_" + "a_questo_momento."); System.out.println(unAltroMelo); unAltroMelo.addGraft(melo); </pre> | <p>Output dell'esempio d'uso:</p> <p>Sono stati creati 2 meli fino a questo momento. Sono stati creati 2 alberi fino a questo momento.</p> <p>tipo: melo innesti: pero</p> <p>Exception in thread "main": java.lang.RuntimeException</p> |
|--|--|

145. (Moto accelerato, 2006-7-17)

Nel contesto di un programma di simulazione per la cinematica, si implementi una classe **Body** che rappresenta un corpo puntiforme dotato di massa, che si sposta lungo una retta. Il costruttore della classe prende come argomento la massa del corpo. Il corpo si suppone inizialmente in quiete alla coordinata 0. Il metodo **setForce** imposta il valore di una forza che viene applicata al corpo. Si supponga che tutte le grandezze siano espresse in unità tra loro omogenee (posizione in metri, velocità in metri al secondo, forza in Newton, etc.). Il metodo **progress** simula il passaggio di un dato numero di secondi, andando ad aggiornare la posizione del corpo. Il metodo **toString** va ridefinito in modo da mostrare la posizione e la velocità corrente del corpo.

Si ricordano le equazioni del moto uniformemente accelerato.

$$F = ma; \quad v = v_0 + at; \quad s = s_0 + v_0 t + \frac{1}{2} a t^2.$$

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre> Body b = new Body(20); b.setForce(40); System.out.println(b); b.progress(1); System.out.println(b); b.progress(2); System.out.println(b); b.setForce(-100); b.progress(2); System.out.println(b); </pre> | <p>Output dell'esempio d'uso:</p> <p>posizione: 0.0, velocita': 0.0 posizione: 1.0, velocita': 2.0 posizione: 9.0, velocita': 6.0 posizione: 11.0, velocita': -4.0</p> |
|--|---|

146. (Moto bidimensionale, 2006-6-26)

Nel contesto di un programma di simulazione per la cinematica, si implementi una classe **Body** che rappresenta un corpo puntiforme dotato di posizione nel piano cartesiano e di velocità. Il costruttore della classe prende come argomento le coordinate alle quali si trova inizialmente il corpo; il corpo si suppone inizialmente in quiete. Il metodo **setSpeed** prende il valore della velocità lungo i due assi di riferimento. Si supponga che la posizione sia espressa in metri e la velocità in metri al secondo. Il metodo **progress** simula il passaggio di un dato numero di secondi, andando ad aggiornare la posizione del corpo. Il metodo **toString** va ridefinito in modo da mostrare la posizione corrente del corpo.

Esempio d'uso:

```
Body b = new Body(0, 0);
```

```
b.setSpeed(1,-1.5);
System.out.println(b);
b.progress(1);
System.out.println(b);
b.progress(2);
System.out.println(b);
```

Output del codice precedente:

```
0.0, 0.0
1.0, -1.5
3.0, -4.5
```

147. (Average, 2006-4-27)

Si implementi una classe **Average** che rappresenti la media aritmetica di un elenco di numeri interi. Ogni oggetto deve possedere un metodo **add** che aggiunge un intero all'elenco, ed un metodo **getAverage** che restituisce la media dei valori immessi fino a quel momento. Il tentativo di chiamare **getAverage** prima che venga inserito alcun valore deve portare ad una eccezione. Esempio di utilizzo:

```
public static void main(String[] x) {
    Average a = new Average();
    double d;

    a.add(10);
    a.add(20);
    d = a.getAverage();
    System.out.println("Media corrente:" + d);

    a.add(60);
    d = a.getAverage();
    System.out.println("Media corrente:" + d);
}
```

Output del codice precedente:

```
Media corrente: 15.0
Media corrente: 30.0
```

Dei 30 punti totali, 10 sono riservati a chi implementa una soluzione che non memorizza tutti gli interi inseriti.

4 Java Collection Framework (collezioni)

148. (Product e Cart, 2020-1-24)

Nell'ambito di un sistema di commercio elettronico, implementare le classi `Product` e `Cart` (carrello della spesa). Un prodotto è caratterizzato da descrizione e prezzo. Il tentativo di istanziare due oggetti prodotto con la stessa descrizione deve produrre un'eccezione.

Un carrello può contenere diversi prodotti, compresi eventuali duplicati, ed offre le seguenti funzionalità:

- Aggiungere un prodotto
- Rimuovere un prodotto
- Conoscere il prezzo totale dei prodotti inseriti

L'implementazione deve rispettare il seguente caso d'uso:

| Esempio d'uso: | Output: |
|---|--|
| <pre>Product sedia = new Product("Sedia_elegante", 100); Product tavolo = new Product("Tavolo_di_cristallo", 200); Cart cart = new Cart(); cart.add(sedia); cart.add(tavolo); cart.add(sedia); System.out.println(cart.totalPrice()); cart.remove(sedia); System.out.println(cart.totalPrice()); Product sedia2 = new Product("Sedia_elegante", 150);</pre> | 400 300 Exception in thread "main"... |

149. (SortedList, 2019-6-24)

Realizzare la classe `SortedList`, che rappresenta una lista di oggetti dotati di ordinamento naturale. Come una normale lista, una `SortedList` accetta duplicati. Inoltre, è iterabile e i suoi iteratori la percorrono in ordine non decrescente.

Nessun metodo di questa classe può avere una complessità superiore a $O(n)$, dove n è la lunghezza della lista.

L'implementazione deve rispettare il seguente esempio d'uso.

| Esempio d'uso: | Output: |
|--|-----------------------|
| <pre>SortedList<Integer> list = new SortedList<>(); list.add(100); list.add(50); list.add(25); list.add(50); for (Integer n: list) System.out.println(n);</pre> | 25 50 50 100 |

Suggerimento: si consideri il metodo `void add(int i, T element)` dell'interfaccia `List<T>`, che inserisce un elemento alla posizione i -esima, spostando tutti gli elementi successivi di una posizione.

150. (RotatingList, 2019-4-29)

[CROWDGRADER] Realizzare la classe `RotatingList`, che rappresenta una lista in grado di ruotare, con le seguenti funzionalità:

- Un costruttore senza argomenti che crea una lista vuota.
- Il metodo `add` per inserire un elemento in coda.

- I metodi `rotateLeft` e `rotateRight` che ruotano la lista di una posizione.
- Un overriding di `equals` che consideri uguali due liste se contengono gli stessi elementi, anche in ordine diverso e in molteplicità diversa (ad esempio, la lista [1, 2, 1] va considerata uguale a [2, 1, 2, 2]).

Nota: le collezioni standard sovrascrivono `toString` in modo da stampare il proprio contenuto.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|---|--|
| Esempio d'uso: | Output: |
| <pre>RotatingList<Integer> l = new RotatingList<>(); l.add(1); l.add(2); l.add(3); System.out.println(l); l.rotateLeft(); System.out.println(l); l.add(4); System.out.println(l); l.rotateRight(); System.out.println(l);</pre> | <pre>[1, 2, 3] [2, 3, 1] [2, 3, 1, 4] [4, 2, 3, 1]</pre> |

151. (Library, 2019-3-19)

Realizzare per una biblioteca le classi `Library` e `Book`. Un oggetto `Book` è caratterizzato dal suo titolo. La classe `Library` offre le seguenti funzionalità:

- Un costruttore senza argomenti che crea una biblioteca vuota.
- Il metodo `addBook` aggiunge un libro alla biblioteca. Se il libro era già stato aggiunto, restituisce `false`.
- Il metodo `loanBook` prende un libro come argomento e lo dà in prestito, a patto che sia disponibile. Se quel libro è già in prestito, restituisce `false`. Se quel libro non è mai stato inserito nella biblioteca, lancia un'eccezione.
- Il metodo `returnBook` prende un libro come argomento e restituisce quel libro alla biblioteca. Se quel libro non era stato prestato col metodo `loanBook`, il metodo `returnBook` lancia un'eccezione.
- Il metodo `printLoans` stampa la lista dei libri attualmente in prestito, *in ordine temporale* (a partire dal libro in prestito da più tempo).

Inoltre, rispondere alla seguente domanda: nella vostra implementazione, qual è la complessità dei metodi `loanBook` e `returnBook`, rispetto al numero di libri n inseriti nella biblioteca?

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|---|
| Esempio d'uso: | Output: |
| <pre>Library lib = new Library(); Book a = new Book("a"), b = new Book("b"), c = new Book("c"); System.out.println(lib.addBook(a)); System.out.println(lib.addBook(b)); System.out.println(lib.addBook(c)); System.out.println(lib.addBook(a)); System.out.println(lib.loanBook(b)); System.out.println(lib.loanBook(a)); lib.printLoans();</pre> | <pre>true true true false true true b a</pre> |

152. (Cellphone, 2018-9-17)

Implementare la classe `Cellphone`, che rappresenta un'utenza telefonica dotata di un gestore (stringa) e un numero di telefono (stringa). La classe è in grado di calcolare il costo di una telefonata, in base al gestore di partenza, al gestore di arrivo, e alla durata.

Il metodo `setCost` consente di impostare il costo al minuto di una telefonata con un dato gestore di partenza e un dato gestore di arrivo. Il metodo `getCost` calcola il costo di una telefonata verso una data utenza e di una data durata (in minuti).

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Cellphone a = new Cellphone("TIMMY", "3341234"),
      b = new Cellphone("Megafon", "3355678"),
      c = new Cellphone("Odissey", "3384343");
Cellphone.setCost("TIMMY", "TIMMY", 0.05);
Cellphone.setCost("TIMMY", "Megafon", 0.15);
Cellphone.setCost("Megafon", "TIMMY", 0.25);

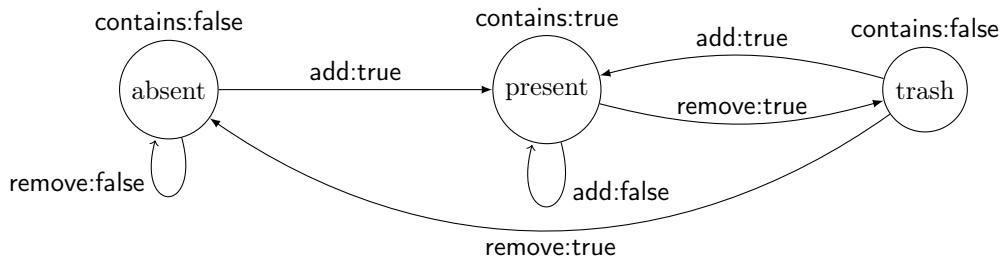
System.out.println(a.getCost(b, 10));
System.out.println(b.getCost(a, 8));
System.out.println(a.getCost(c, 10)); // provoca eccezione
```

Output:

```
1.5
2.0
Exception in
thread "main" ...
```

153. (SafeSet, 2018-7-19)

Realizzare la classe `SafeSet`, che rappresenta un insieme che richiede due passaggi per rimuovere completamente un oggetto. Il metodo `add` aggiunge un elemento all'insieme, restituendo `true` se l'inserimento ha avuto successo. Il metodo `remove` rimuove un elemento dall'insieme, ma la rimozione è definitiva solo dopo una seconda chiamata. Il metodo `contains` verifica se l'insieme contiene un dato elemento (in base a `equals`). Infine, un `SafeSet` deve essere *thread-safe*. Il seguente diagramma rappresenta il ciclo di vita di un oggetto all'interno di un `SafeSet`:



L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
SafeSet<String> a = new SafeSet<>();
System.out.println(a.add("ciao"));
System.out.println(a.add("mondo"));
System.out.println(a.remove("ciao"));
System.out.println(a.contains("ciao"));
System.out.println(a.remove("ciao"));
System.out.println(a.contains("ciao"));
```

Output:

```
true
true
true
false
true
false
```

154. (Merge, 2018-5-2)

Realizzare un metodo chiamato `merge` che rispetti il seguente contratto:

Pre-condizione Accetta due `LinkedList` dello stesso tipo e di pari lunghezza.

Post-condizione Restituisce una nuova `LinkedList` ottenuta alternando gli elementi della prima lista e quelli della seconda.

Ad esempio, se la prima lista contiene (1, 2, 3) e la seconda lista (4, 5, 6), la nuova lista deve contenere (1, 4, 2, 5, 3, 6).

Penale Se le liste non hanno la stessa lunghezza, lancia `IllegalArgumentException`.

155. (isSetSmaller, 2018-3-23)

Implementare il metodo statico `isSetSmaller`, che accetta due insiemi e un comparatore, e restituisce vero se e solo se tutti gli elementi del primo insieme sono più piccoli, in base al comparatore, di tutti gli elementi del secondo insieme.

Porre particolare attenzione alla scelta della firma.

156. (Book e Library, 2018-2-22)

Realizzare le classi `Book` e `Library`, che rappresentano rispettivamente un libro e una collezione di libri. Il metodo `addBook` di `Library` aggiunge un libro alla collezione, con un dato titolo e un dato autore. A ciascun libro è possibile attribuire uno o più argomenti tramite il suo metodo `addTag`. Il metodo `getBooksByTag` di `Library` restituisce *in tempo costante* l'insieme dei libri di un argomento dato.

L'implementazione deve rispettare il seguente esempio d'uso:

Esempio d'uso:

```
Library casa = new Library(), ufficio = new Library();

Library.Book b1 = casa.addBook("Esercizi_di_stile", "Queneau");
b1.addTag("letteratura");
b1.addTag("umorismo");
Library.Book b2 = casa.addBook("Me_parlare_bene_un_giorno", "Sedaris");
b2.addTag("umorismo");
Library.Book b3 = ufficio.addBook("Literate_programming", "Knuth");
b3.addTag("programmazione");

Set<Library.Book> humorCasa = casa.getBooksByTag("umorismo");
System.out.println(humorCasa);
Set<Library.Book> humorUfficio = ufficio.getBooksByTag("umorismo");
System.out.println(humorUfficio);
```

Output:

```
[Esercizi di stile, by Queneau, Me parlare bene un giorno, by Sedaris]
null
```

157. (Component e Configuration, 2018-10-18)

Un sito vuole consentire agli utenti di ordinare computer assemblati. Data l'enumerazione:

```
enum Type { CPU, BOARD, RAM; }
```

implementare le classi `Component`, che rappresenta un componente di un PC, e `Configuration`, che rappresenta un PC da assemblare.

Un componente è caratterizzato dalla sua tipologia (`Type`) e da una descrizione (stringa). Il suo metodo `setIncompatible` dichiara che questo componente è incompatibile con un altro componente, passato come argomento. Un componente può essere incompatibile con diversi altri componenti.

Il metodo `add` di `Configuration` aggiunge un componente a questo PC e restituisce `true`, ma solo se il componente è compatibile con quelli già inseriti, ed è di tipo diverso da quelli già inseriti, altrimenti non lo inserisce e restituisce `false`.

Suggerimento: Una classe `Component` ben progettata non nominerà le 3 istanze di `Type`.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre> Component cpu1 = new Component(Type.CPU, "Ryzen_5_2600"), cpu2 = new Component(Type.CPU, "Core_i5_7500"), board1 = new Component(Type.BOARD, "Prime_X470"), board2 = new Component(Type.BOARD, "Prime_Z370"), ram = new Component(Type.RAM, "DDR4_8GB"); cpu1.setIncompatible(board2); board1.setIncompatible(cpu2); Configuration pc = new Configuration(); System.out.println(pc.add(cpu1)); System.out.println(pc.add(cpu2)); // due cpu! System.out.println(pc.add(board2)); // incompatible! System.out.println(pc.add(board1)); System.out.println(pc.add(ram)); </pre> | <p>Output:</p> <pre> true false false true true </pre> |
|--|--|

158. (Bug, 2018-1-24)

Realizzare la classe Bug, che rappresenta un errore in un programma. Il costruttore accetta una descrizione dell'errore. Inizialmente, l'errore non è assegnato ad alcuno sviluppatore. Il metodo assignTo assegna l'errore ad uno sviluppatore, identificato dal nome, che sarà incaricato di risolvere l'errore.

Il metodo statico getUnassigned restituisce *in tempo costante* l'insieme degli errori non ancora assegnati. Il metodo statico getAssignedTo restituisce *in tempo costante* l'insieme degli errori assegnati ad uno sviluppatore dato.

Nota: un bug assegnato ad uno sviluppatore può essere riassegnato ad un altro.

L'implementazione deve rispettare il seguente esempio d'uso:

| | |
|---|--|
| <p>Esempio d'uso:</p> <pre> Bug b1 = new Bug("Application_crashes_on_Windows_8"), b2 = new Bug("Application_freezes_after_2_hours"), b3 = new Bug("Application_does_not_print_on_laser_ printer"), b4 = new Bug("Data_missing_after_partial_save"); Set<Bug> unassigned = Bug.getUnassigned(); System.out.println(unassigned.size()); b2.assignTo("Paolo"); b3.assignTo("Filomena"); b4.assignTo("Filomena"); System.out.println(unassigned.size()); Set<Bug> filo = Bug.getAssignedTo("Filomena"); System.out.println(filo); </pre> | <p>Output:</p> <pre> 4 1 [("Data missing after partial save", assigned to Filomena), ("Application does not print on laser printer", assigned to Filomena)] </pre> |
|---|--|

159. (Room, 2017-4-26)

[CROWDGRADER] Realizzare le classi Room e Reservation, che rappresentano una camera d'albergo e una prenotazione per la camera. Il metodo reserve di Room accetta un nome, la data di inizio e di fine prenotazione, e restituisce un oggetto di tipo Reservation. Se la camera è occupata in una delle giornate richieste, il metodo lancia un'eccezione. Per semplicità, una data è rappresentata da un numero intero tra 0 a 365. Il metodo reservations di Room consente di scorrere l'elenco delle prenotazioni, in ordine cronologico.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre>Room r = new Room(); Reservation p1 = r.reserve("Pasquale_Cafiero", 105, 120); Reservation p2 = r.reserve("Carlo_Martello", 5, 20); Reservation p3 = r.reserve("Piero", 20, 22); Reservation p4 = r.reserve("Marinella", 200, 222); for (Reservation p: r.reservations()) System.out.println(p.getName());</pre> | <p>Output:</p> <pre>Carlo Martello Piero Pasquale Cafiero Marinella</pre> |
|--|---|

160. (BinRel, 2017-3-23)

Realizzare la classe `BinRel`, che rappresenta una relazione binaria tra un insieme e se stesso. Il metodo `addPair` aggiunge una coppia di oggetti alla relazione. Il metodo `areRelated` verifica se una data coppia di oggetti appartiene alla relazione. Il metodo `isSymmetric` verifica se la relazione è simmetrica.

L'implementazione deve rispettare il seguente caso d'uso:

| | |
|---|---|
| <p>Esempio d'uso:</p> <pre>BinRel<String> rel = new BinRel<>(); rel.addPair("a", "albero"); rel.addPair("a", "amaca"); System.out.println(rel.isSymmetric()); rel.addPair("albero", "a"); rel.addPair("amaca", "a"); System.out.println(rel.isSymmetric()); System.out.println(rel.areRelated("a", "amaca"));</pre> | <p>Output:</p> <pre>false true true</pre> |
|---|---|

161. (Clinica, 2017-10-6)

Data la seguente enumerazione:

```
enum Specialista { OCULISTA, PEDIATRA; }
```

Realizzare la classe `Clinica`, che permette di prenotare e cancellare visite mediche. I metodi `prenota` e `cancellaPrenotazione` accettano uno specialista e il nome di un paziente, ed effettuano o cancellano la prenotazione, rispettivamente. Il metodo `getPrenotati` restituisce l'elenco dei prenotati.

La classe deve rispettare le seguenti proprietà:

- a) Non ci si può prenotare con più di uno specialista.
- b) Si deve poter aggiungere uno specialista all'enumerazione senza dover modificare la classe `Clinica`.

Inoltre, l'implementazione deve rispettare il seguente esempio d'uso:

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre>Clinica c = new Clinica(); c.prenota(Specialista.OCULISTA, "Pippo_Franco"); c.prenota(Specialista.OCULISTA, "Leo_Gullotta"); c.prenota(Specialista.OCULISTA, "Leo_Gullotta"); c.prenota(Specialista.PEDIATRA, "Ciccio_Ingrassia"); c.prenota(Specialista.PEDIATRA, "Leo_Gullotta"); c.cancellaPrenotazione(Specialista.PEDIATRA, "Ciccio_Ingrassia"); Collection<String> ocu = c.getPrenotati(Specialista.OCULISTA); System.out.println(ocu); System.out.println(c.getPrenotati(Specialista.PEDIATRA));</pre> | <p>Output:</p> <pre>[Leo Gullotta, Pippo Franco] []</pre> |
|--|---|

162. (**mergeIfSorted**, 2017-1-25)

Implementare il metodo statico `mergeIfSorted`, che accetta due liste *a* e *b*, e un comparatore *c*, e restituisce un'altra lista. Inizialmente, usando due thread diversi, il metodo verifica che le liste *a* e *b* siano ordinate in senso non decrescente (ogni thread si occupa di una lista). Poi, se le liste sono effettivamente ordinate, il metodo le fonde (senza modificarle) in un'unica lista ordinata, che viene restituita al chiamante. Se, invece, almeno una delle due liste non è ordinata, il metodo termina restituendo `null`.

Il metodo dovrebbe avere complessità di tempo lineare.

Porre particolare attenzione alla scelta della firma, considerando i criteri di funzionalità, completezza, correttezza, fornitura di garanzie e semplicità.

163. (**SocialUser**, 2016-9-20)

Per un social network, implementare le classi `SocialUser` e `Post`. Un utente è dotato di un nome e può creare dei post tramite il metodo `newPost`. Il contenuto di un post è una stringa, che può contenere nomi di utenti, preceduti dal simbolo “@”. Il metodo `getTagged` della classe `Post` restituisce l'insieme degli utenti il cui nome compare in quel post, mentre il metodo `getAuthor` restituisce l'autore del post.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
SocialUser adriana = new SocialUser("Adriana");
barbara = new SocialUser("Barbara");
SocialUser.Post p = adriana.newPost("Ecco_una_foto_con_
@Barbara_e_@Carla.");
Set<SocialUser> tagged = p.getTagged();
System.out.println(tagged);
System.out.println(tagged.iterator().next() == barbara);
System.out.println(p.getAuthor());
```

Output:

```
[Barbara]
true
Adriana
```

Suggerimento: l'invocazione `a.lastIndexOf(b)` restituisce -1 se la stringa *b* *non* è presente nella stringa *a*, e un numero maggiore o uguale di zero altrimenti.

164. (**GameLevel**, 2016-3-3)

Implementare la classe `GameLevel`, che rappresenta un livello in un gioco 2D, in cui un personaggio si muove su una griglia di caselle. Il costruttore accetta le dimensioni del livello (larghezza e altezza). Il metodo `setWall` accetta le coordinate di una casella e mette un muro in quella casella. Il metodo `areConnected` accetta le coordinate di due caselle e restituisce *vero* se e solo se esiste un percorso tra di loro.

Caso d'uso:

```
GameLevel map = new GameLevel(3, 3);
System.out.println(map.areConnected(0,0,2,2));
map.setWall(0,1);
map.setWall(1,1);
System.out.println(map.areConnected(0,0,2,2));
map.setWall(2,1);
System.out.println(map.areConnected(0,0,2,2));
```

Output:

```
true true false
```

165. (**Curriculum**, 2016-1-27)

Un oggetto `Curriculum` rappresenta una sequenza di lavori, ognuno dei quali è un'istanza della classe `Job`. Il costruttore di `Curriculum` accetta il nome di una persona. Il metodo `addJob` aggiunge un lavoro alla sequenza, caratterizzato da una descrizione e dall'anno di inizio, restituendo un nuovo oggetto di tipo `Job`. Infine, la classe `Job` offre il metodo `next`, che restituisce *in tempo costante* il lavoro successivo nella sequenza (oppure `null`).

Implementare le classi `Curriculum` e `Job`, rispettando il seguente caso d'uso.

Caso d'uso:

```
Curriculum cv = new Curriculum("Walter_White");
Curriculum.Job j1 = cv.addJob("Chimico", 1995);
Curriculum.Job j2 = cv.addJob("Insegnante", 2005);
Curriculum.Job j3 = cv.addJob("Cuoco", 2009);

System.out.println(j2.next());
System.out.println(j3.next());
```

Output:

```
Cuoco: 2009
null
```

166. (Progression, 2015-9-21)

Nell'ambito di un programma di gestione del personale, la classe Progression calcola il salario dei dipendenti, in base alla loro anzianità in servizio. Il salario mensile parte da un livello base ed ogni anno solare aumenta di un certo incremento. Il costruttore accetta il salario base e l'incremento annuale. Il metodo addEmployee aggiunge un impiegato a questa progressione, specificando il nome e l'anno di assunzione. Il metodo getSalary restituisce il salario mensile di un impiegato in un dato anno. Infine, il metodo addBonus attribuisce ad un impiegato un bonus extra in un dato anno. Cioè, addBonus("Pippo", 2010, 50) significa che Pippo percepirà 50 euro in più in ogni mese del 2010.

Caso d'uso:

```
Progression a = new Progression(1000, 150);

a.addEmployee("Jesse", 2008);
a.addEmployee("Gale", 2009);
a.addBonus("Gale", 2010, 300);

System.out.println(a.getSalary("Jesse", 2009));
System.out.println(a.getSalary("Gale", 2010));
System.out.println(a.getSalary("Gale", 2011));
```

Output:

```
1150
1450
1300
```

167. (Controller, 2015-6-24)

Realizzare la classe Controller, che rappresenta una centralina per autoveicoli, e la classe Function, che rappresenta una funzionalità del veicolo, che può essere accesa o spenta. Alcune funzionalità sono *incompatibili* tra loro, per cui accenderne una fa spegnere automaticamente l'altra.

La classe Controller ha due metodi: addFunction aggiunge al sistema una nuova funzionalità con un dato nome; printOn stampa a video i nomi delle funzionalità attive. La classe Function ha tre metodi: turnOn e turnOff per attivarla e disattivarla; setIncompatible accetta un'altra funzionalità *x* e imposta un'incompatibilità tra *this* e *x*.

Leggere attentamente il seguente caso d'uso, che mostra, tra le altre cose, che l'incompatibilità è automaticamente simmetrica, ma *non* transitiva.

Caso d'uso:

```
Controller c = new Controller();
Controller.Function ac = c.addFunction("Aria_condizionata");
Controller.Function risc = c.addFunction("Riscaldamento");
Controller.Function sedile = c.addFunction("Sedile_riscaldato");

ac.setIncompatible(risc);
ac.setIncompatible(sedile);

ac.turnOn();
c.printOn();
System.out.println("----");

risc.turnOn();
sedile.turnOn();
c.printOn();
```

Output:

```
Aria condizionata
-----
Sedile riscaldato
Riscaldamento
```

168. (Relation, 2015-1-20)

Realizzare la classe Relation, che rappresenta una relazione binaria tra un insieme S e un insieme T. In pratica, una Relation è analoga ad una Map, con la differenza che la Relation accetta chiavi duplicate.

Il metodo **put** aggiunge una coppia di oggetti alla relazione. Il metodo **remove** rimuove una coppia di oggetti dalla relazione. Il metodo **image** accetta un oggetto x di tipo S e restituisce l'insieme degli oggetti di tipo T che sono in relazione con x. Il metodo **preImage** accetta un oggetto x di tipo T e restituisce l'insieme degli oggetti di tipo S che sono in relazione con x.

Esempio d'uso:

```
Relation<Integer,String> r = new Relation<Integer,String>();
r.put(0, "a"); r.put(0, "b"); r.put(0, "c");
r.put(1, "b"); r.put(2, "c");
r.remove(0, "a");
Set<String> set0 = r.image(0);
Set<Integer> setb = r.preImage("b");
System.out.println(set0);
System.out.println(setb);
```

Output:

```
[b, c]
[0, 1]
```

169. (Contest, 2014-9-18)

Un oggetto di tipo Contest consente ai client di votare per uno degli oggetti che partecipano a un “concorso”. Implementare la classe parametrica Contest con i seguenti metodi: il metodo **add** consente di aggiungere un oggetto al concorso; il metodo **vote** permette di votare per un oggetto; se l'oggetto passato a **vote** non partecipa al concorso (cioè non è stato aggiunto con **add**), viene lanciata un'eccezione; il metodo **winner** restituisce uno degli oggetti che fino a quel momento ha ottenuto più voti.

Tutti i metodi devono funzionare in tempo costante.

| | |
|---|----------------|
| Esempio d'uso: | Output: Red |
| <pre>Contest<String> c = new Contest<String>(); String r = "Red", b = "Blue", g = "Green"; c.add(r); c.vote(r); c.add(b); c.add(g); c.vote(r); c.vote(b); System.out.println(c.winner());</pre> | |

170. (**subMap**, 2014-7-3)

Implementare il metodo **subMap** che accetta una mappa e una collezione e restituisce una nuova mappa ottenuta restringendo la prima alle sole chiavi che compaiono nella collezione. Il metodo non modifica i suoi argomenti.

Valutare le seguenti intestazioni per il metodo **subMap**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K> Map<K,?> subMap(Map<K,?> m, Collection<K> c)
- b) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<?> c)
- c) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<? super K> c)
- d) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<? extends K> c)
- e) <K,V> Map<K,V> subMap(Map<K,V> m, Set<K> c)
- f) <K,V,K2 extends K> Map<K,V> subMap(Map<K,V> m, Collection<K2> c)

171. (**inverseMap**, 2014-7-28)

Implementare il metodo **inverseMap** che accetta una mappa **m** e ne restituisce una nuova, ottenuta invertendo le chiavi con i valori. Se **m** contiene valori duplicati, il metodo lancia un'eccezione. Il metodo non modifica la mappa **m**.

Valutare le seguenti intestazioni per il metodo **inverseMap**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K,V> Map<V,K> inverseMap(Map<?,?> m)
- b) Map<?,?> inverseMap(Map<?,?> m)
- c) <K,V> Map<K,V> inverseMap(Map<V,K> m)
- d) <K,V> Map<K,V> inverseMap(Map<? extends V, ? super K> m)
- e) <K,V> Map<K,V> inverseMap(Map<K,V> m)
- f) <K,V> Map<K,V> inverseMap(Map<? extends V, ? extends K> m)

172. (**BoundedSet**, 2014-1-31)

Realizzare la classe **BoundedSet**, che rappresenta un insieme di capacità limitata. Il costruttore accetta la capacità massima dell'insieme. La classe deve implementare i metodi **add**, **contains** e **size** secondo il contratto previsto dall'interfaccia **Set**. Se però l'insieme è alla sua capacità massima e si tenta di inserire un nuovo elemento con **add**, prima dell'inserimento sarà cancellato dall'insieme l'elemento che vi è stato inserito prima (cioè, l'elemento più “vecchio” presente nell'insieme).

Fare in modo che sia **add** sia **contains** funzionino in tempo costante.

| | |
|---|--------------------------------------|
| Esempio d'uso: <pre>BoundedSet<Integer> s = new BoundedSet<Integer>(3); s.add(3); s.add(8); s.add(5); s.add(5); System.out.println(s.size()); System.out.println(s.contains(3)); s.add(14); System.out.println(s.size()); System.out.println(s.contains(3));</pre> | Output: <pre>3 true 3 false</pre> |
|---|--------------------------------------|

173. (Movie, 2013-9-25)

La classe Movie rappresenta un film, con attributi titolo (stringa) e anno di produzione (intero). Alcuni film formano delle serie, cioè sono dei *sequel* di altri film. La classe offre due costruttori: uno per film normali e uno per film appartenenti ad una serie. Quest'ultimo costruttore accetta come terzo argomento il film di cui questo è il successore.

Il metodo `getSeries` restituisce la lista dei film che formano la serie a cui questo film appartiene. Se invocato su un film che non appartiene ad una serie, il metodo restituisce una lista contenente solo questo film.

Il metodo statico `selectByYear` restituisce l'insieme dei film prodotti in un dato anno, in tempo costante.

| | |
|--|---|
| Esempio d'uso: <pre>Movie r1 = new Movie("Rocky", 1976); Movie r2 = new Movie("Rocky_II", 1979, r1); Movie r3 = new Movie("Rocky_III", 1982, r2); Movie f = new Movie("Apocalypse_Now", 1979); Set<Movie> movies1979 = Movie.selectByYear(1979); System.out.println(movies1979); List<Movie> rockys = r2.getSeries(); System.out.println(rockys);</pre> | Output: <pre>[Rocky II, Apocalypse Now] [Rocky, Rocky II, Rocky III]</pre> |
|--|---|

174. (composeMaps, 2013-9-25)

Il metodo `composeMaps` accetta due mappe *a* e *b*, e restituisce una nuova mappa *c* così definita: le chiavi di *c* sono le stesse di *a*; il valore associato in *c* ad una chiave *x* è pari al valore associato nella mappa *b* alla chiave *a(x)*.

Nota: Se consideriamo le mappe come funzioni matematiche, la mappa c è definita come $c(x) = b(a(x))$, cioè come composizione di a e b.

Valutare ciascuna delle seguenti intestazioni per il metodo `composeMaps`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, specificità del tipo di ritorno e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<T, U> b)
- b) <S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<? extends T, U> b)
- c) <S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<? super T, U> b)
- d) <S, U> Map<S,U> composeMaps(Map<S, ?> a, Map<?, U> b)
- e) <S, U> Map<S,U> composeMaps(Map<S, Object> a, Map<Object, U> b)

175. (isSorted, 2013-7-9)

Implementare il metodo `isSorted` che accetta una lista e un comparatore, e restituisce `true` se la lista risulta già ordinata in senso non-decrescente rispetto a quel comparatore, e `false` altrimenti.

Valutare ciascuna delle seguenti intestazioni per il metodo `isSorted`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) **boolean** `isSorted(List<?> x, Comparator<Object> c)`
- b) `<S> boolean isSorted(List<? extends S> x, Comparator<? super S> c)`
- c) `<S> boolean isSorted(List<S> x, Comparator<S> c)`
- d) **boolean** `isSorted(List<Object> x, Comparator<Object> c)`
- e) `<S, T> boolean isSorted(List<S> x, Comparator<T> c)`
- f) `<S, T extends S> boolean isSorted(List<T> x, Comparator<S> c)`

176. (MultiBuffer, 2013-6-25)

Implementare la classe parametrica `MultiBuffer`, che rappresenta un insieme di buffer. Il suo costruttore accetta il numero n di buffer da creare. Il metodo `insert` inserisce un oggetto nel buffer che in quel momento contiene meno elementi. Il metodo bloccante `take` accetta un intero i compreso tra 0 ed $n - 1$ e restituisce il primo oggetto presente nel buffer i -esimo. La classe deve risultare *thread-safe*.

| Esempio d'uso: | Output: |
|--|---------|
| <pre>MultiBuffer<Integer> mb = new MultiBuffer<Integer>(3); mb.insert(13); mb.insert(24); mb.insert(35); System.out.println(mb.take(0));</pre> | 13 |

Si consideri il seguente schema di sincronizzazione: `insert` è mutuamente esclusivo con `take(i)`, per ogni valore di i ; `take(i)` è mutuamente esclusivo con `take(j)`, ma è compatibile con `take(j)`, quando j è diverso da i . Rispondere alle seguenti domande:

- a) Questo schema evita le *race condition*?
- b) E' possibile implementare questo schema utilizzando metodi e blocchi sincronizzati?

177. (Concat, 2013-6-25)

Implementare il metodo `concat` che accetta due iteratori a e b e restituisce un altro iteratore che restituisce prima tutti gli elementi restituiti da a e poi tutti quelli di b .

Valutare le seguenti intestazioni per il metodo `concat`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `Iterator<Object> concat(Iterator<Object> a, Iterator<Object> b)`
- b) `Iterator<?> concat(Iterator<?> a, Iterator<?> b)`
- c) `<S> Iterator<S> concat(Iterator<S> a, Iterator<S> b)`
- d) `<S> Iterator<S> concat(Iterator<? extends S> a, Iterator<? extends S> b)`
- e) `<S,T> Iterator<S> concat(Iterator<S> a, Iterator<T> b)`
- f) `<S,T extends S> Iterator<T> concat(Iterator<T> a, Iterator<S> b)`

178. (City, 2013-4-29)

La classe `City` rappresenta una città. Il costruttore accetta il nome della città, mentre il metodo `connect` accetta un'altra città e stabilisce un collegamento tra le due (una strada o un altro tipo di collegamento). Tutti i collegamenti sono bidirezionali.

Il metodo `getConnections` restituisce la collezione delle città *direttamente* collegate a questa. Il metodo `isConnected` prende come argomento un'altra città e restituisce vero se è collegata a `this` *direttamente* o *indirettamente* (cioè, tramite un numero arbitrario di collegamenti).

| | |
|---|--------------------------------------|
| Esempio d'uso: <pre>City n = new City("Napoli"), r = new City("Roma"), s = new City ("Salerno"), p = new City("Parigi"); n.connect(s); n.connect(r); Collection<City> r_conn = r.getConnections(); System.out.println(r_conn); System.out.println(r.isConnected(s)); System.out.println(r.isConnected(p));</pre> | Output: [Napoli] true false |
|---|--------------------------------------|

179. (Auditorium, 2013-3-22)

La seguente classe (semplificata) **Seat** rappresenta un posto in un auditorium.

```
public class Seat { public int row, col; }
```

La classe Auditorium serve ad assegnare i posti in un teatro. Il costruttore prende come argomenti le dimensioni della platea, in termini di file e posti per fila, nonché un oggetto **Comparator** che serve ad ordinare i posti in ordine di preferenza. Il metodo **assignSeats** prende come argomenti il numero *n* di posti richiesti e restituisce un insieme contenente gli *n* posti migliori (in base al comparatore) ancora disponibili. Se la platea non contiene *n* posti disponibili, il metodo restituisce **null**.

| | |
|--|--------------------------------------|
| Esempio d'uso: <pre>Auditorium a = new Auditorium(5, 5, new Comparator<Seat>() { public int compare(Seat a, Seat b) { return (a.row==b.row)? (a.col-b.col): (a.row-b.row); } }); Set<Seat> s = a.assignSeats(4); System.out.println(s);</pre> | Output: [(0,0),(0,1),(0,2),(0,3)] |
|--|--------------------------------------|

180. (MultiSet, 2013-2-11)

Un MultiSet è un insieme in cui ogni elemento può comparire più volte. Quindi, ammette duplicati come una lista, ma, a differenza di una lista, l'ordine in cui gli elementi vengono inseriti non è rilevante. Implementare una classe parametrica **MultiSet**, con i seguenti metodi:

- **add**, che aggiunge un elemento,
- **remove**, che rimuove un elemento (se presente), ed
- **equals**, che sovrascrive quello di **Object** e considera uguali due MultiSet se contengono gli stessi elementi, ripetuti lo stesso numero di volte.

Infine, deve essere possibile iterare su tutti gli elementi di un MultiSet usando il ciclo **for-each**.

| | |
|--|--|
| Esempio d'uso: <pre>MultiSet<Integer> s1 = new MultiSet<Integer>(); MultiSet<Integer> s2 = new MultiSet<Integer>(); s1.add(5); s1.add(7); s1.add(5); s2.add(5); s2.add(5); s2.add(7); for (Integer n: s1) System.out.println(n); System.out.println(s1.equals(s2));</pre> | Output (l'ordine dei numeri è irrilevante): 7 5 5 true |
|--|--|

181. (Bijection, 2012-9-3)

Implementare la classe parametrica **Bijection**, che rappresenta una biezione (relazione biunivoca) tra un insieme di chiavi e uno di valori. Il metodo **addPair** aggiunge una coppia chiave-valore alla relazione. Se la chiave oppure il valore sono già presenti, viene lanciata un'eccezione. Il metodo **getValue** accetta come argomento una chiave e restituisce il valore associato, oppure **null** se la chiave non è presente. Il metodo **getKey** accetta un valore e restituisce la chiave corrispondente, oppure **null** se il valore non è presente. Entrambi i metodi **getValue** e **getKey** devono operare in tempo costante.

| | |
|--|---|
| Esempio d'uso: | Output: |
| <pre>Bijection<Integer,String> b = new Bijection<Integer,String>(); b.addPair(12,"dodici"); b.addPair(7,"sette"); System.out.println(b.getValue(12)); System.out.println(b.getKey("tredici")); b.addPair(8,"sette");</pre> | dodici null Exception in thread "main" ... |

182. (Social network, 2012-7-9)

Nell'ambito dell'implementazione di un *social network*, la classe **Person** rappresenta un utente. Tramite i metodi **addFriend** e **addEnemy** è possibile aggiungere un amico o un nemico a questa persona, rispettando le seguenti regole:

- a) una persona non può aggiungere se stessa come amico o nemico;
- b) una persona non può aggiungere la stessa persona sia come amico sia come nemico.

Il metodo **contacts** permette di iterare su tutti i contatti di questa persona tramite un iteratore, che restituirà prima tutti gli amici e poi tutti i nemici.

| | |
|---|--------------------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Person a = new Person("Antonio"); Person c = new Person("Cleopatra"); Person o = new Person("Ottaviano"); a.addEnemy(o); a.addFriend(c); for (Person p: a.contacts()) System.out.println(p);</pre> | Cleopatra Ottaviano |

183. (BoundedMap, 2012-6-18)

Implementare la classe **BoundedMap**, che rappresenta una mappa con capacità limitata. Il costruttore accetta la dimensione massima della mappa. I metodi **get** e **put** sono analoghi a quelli dell'interfaccia **Map**. Se però la mappa è piena e viene invocato il metodo **put** con una chiave nuova, verrà rimossa dalla mappa la chiave che fino a quel momento è stata ricercata meno volte con **get**.

L'implementazione deve rispettare il seguente caso d'uso.

| | |
|--|------------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>BoundedMap<String,String> m = new BoundedMap<String,String>(2); m.put("NA", "Napoli"); m.put("SA", "Salerno"); System.out.println(m.get("NA")); m.put("AV", "Avellino"); System.out.println(m.get("SA"));</pre> | Napoli null |

184. (Panino, 2012-4-23)

Implementare la classe **Panino**, il cui metodo **addIngrediente** aggiunge un ingrediente, scelto da un elenco fisso. Gli ingredienti sono divisi in categorie. Se si tenta di aggiungere più di un ingrediente della stessa categoria, il metodo **addIngrediente** solleva un'eccezione.

Elenco delle categorie e degli ingredienti:

ripieni: PROSCIUTTO, SALAME

formaggi: SOTTILETTA, MOZZARELLA

salse: MAIONESE, SENAPE

| | |
|---|--|
| <p>Esempio d'uso:</p> <pre>Panino p = new Panino(); p.addIngrediente(Panino.Ingrediente.SALAME); p.addIngrediente(Panino.Ingrediente.SOTTILETTA); System.out.println(p); p.addIngrediente(Panino.Ingrediente.MOZZARELLA);</pre> | <p>Output dell'esempio d'uso: panino con SALAME, SOTTILETTA Exception in thread "main"...</p> |
|---|--|

185. (MakeMap, 2011-2-7)

Scrivere un metodo che accetta due liste (List) k e v di pari lunghezza, e restituisce una mappa in cui all'elemento i -esimo di k viene associato come valore l'elemento i -esimo di v .

Il metodo lancia un'eccezione se le liste non sono di pari lunghezza, oppure se k contiene elementi duplicati.

Si ricordi che non è opportuno utilizzare l'accesso posizionale su liste generiche.

186. (Intersect, 2010-9-14)

Implementare il metodo statico `intersect`, che accetta come argomenti due Collection x e y e restituisce una nuova Collection che contiene l'intersezione di x ed y (cioè, gli oggetti comuni ad entrambe le collezioni).

Prestare particolare attenzione alla scelta della firma del metodo.

187. (PartiallyComparable, 2010-6-28)

L'interfaccia `PartComparable` (per *partially comparable*) rappresenta un tipo i cui elementi sono *parzialmente* ordinati.

```
public interface PartComparable<T> {
    public PartComparison compareTo(T x);
}
public enum PartComparison {
    SMALLER, EQUAL, GREATER, UNCOMP;
}
```

Implementare la classe `POSet` (per *partially ordered set*), che rappresenta un insieme parzialmente ordinato, i cui elementi implementano l'interfaccia `PartComparable`. Un oggetto di questo insieme è detto *massimale* se nessun altro oggetto nell'insieme è maggiore di lui.

Il metodo `add` aggiunge un oggetto all'insieme, mentre il metodo `isMaximal` restituisce vero se l'oggetto passato come argomento è uno degli oggetti massimali dell'insieme, restituisce falso se tale oggetto appartiene all'insieme, ma non è massimale, ed infine solleva un'eccezione se l'oggetto non appartiene all'insieme. Il metodo `isMaximal` deve terminare in tempo costante.

| | |
|---|--|
| <pre>// Stringhe, ordinate parzialmente dalla relazione di prefisso class POString implements PartComparable<POString> { ... } POSet<POString> set = new POSet<POString>(); set.add(new POString("architetto")); set.add(new POString("archimede")); set.add(new POString("archi")); set.add(new POString("bar")); set.add(new POString("bari")); System.out.println(set.isMaximal(new POString("archimede"))) ; System.out.println(set.isMaximal(new POString("bar"))); set.add(new POString("archimedeo")); System.out.println(set.isMaximal(new POString("archimede"))) ;</pre> | <p>Output dell'esempio d'uso: true false false</p> |
|---|--|

188. (SelectKeys, 2010-11-30)

Scrivere un metodo che accetta una lista l e una mappa m , e restituisce una nuova lista che contiene gli elementi di l che compaiono come chiavi in m . Porre particolare attenzione alla scelta della firma.

189. (Color, 2010-1-22)

La classe `Color` rappresenta un colore, determinato dalle sue componenti RGB. La classe offre alcuni colori predefiniti, tra cui `RED`, `GREEN` e `BLUE`. Un colore nuovo si può creare solo con il metodo factory `make`. Se il client cerca di ricreare un colore predefinito, gli viene restituito quello e non uno nuovo. Ridefinire anche il metodo `toString`, in modo che rispetti il seguente caso d'uso.

| Esempio d'uso: | Output dell'esempio d'uso: |
|---|---|
| <pre>Color rosso = Color.RED; Color giallo = Color.make(255, 255, 0); Color verde = Color.make(0, 255, 0); System.out.println(rosso); System.out.println(giallo); System.out.println(verde); System.out.println(verde == Color. GREEN);</pre> | <pre>red (255, 255, 0) green true</pre> |

190. (GetByType, 2010-1-22)

Implementare il metodo statico `getByType` che, data una collezione c (`Collection`) ed un oggetto x di tipo `Class`, restituisce un oggetto della collezione il cui tipo effettivo sia esattamente x . Se un tale oggetto non esiste, il metodo restituisce `null`.

Prestare particolare attenzione alla scelta della firma del metodo. Si ricordi che la classe `Class` è parametrica.

191. (Tutor, 2009-6-19)

Un *tutor* è un dispositivo per la misurazione della velocità media in autostrada. Una serie di sensori identifica i veicoli in base alle targhe e ne calcola la velocità, misurando il tempo che il veicolo impiega a passare da un sensore al successivo (e, naturalmente, conoscendo la distanza tra i sensori).

Si implementi la classe `Tutor` e la classe `Detector` (sensore). Il metodo `addDetector` di `Tutor` crea un nuovo sensore posto ad un dato chilometro del tracciato. Il metodo `carPasses` di `Detector` rappresenta il passaggio di un veicolo davanti a questo sensore: esso prende come argomenti la targa di un veicolo ed un tempo assoluto in secondi, e restituisce una stima della velocità di quel veicolo, basata anche sui dati dei sensori che lo precedono. Tale metodo restituisce -1 se il sensore non ha sufficienti informazioni per stabilire la velocità.

Si supponga che le chiamate ad `addDetector` avvengano tutte all'inizio e con chilometri crescenti, come nel seguente esempio d'uso.

| Esempio d'uso: | Output dell'esempio d'uso: |
|---|----------------------------|
| <pre>Tutor tang = new Tutor(); Tutor.Detector a = tang.addDetector(2); Tutor.Detector b = tang.addDetector(5); Tutor.Detector c = tang.addDetector(9); // nuovo veicolo System.out.println(a.carPasses("NA12345", 0)); // 3km in 1200 sec (20 minuti), quindi 9km/h System.out.println(b.carPasses("NA12345", 1200)); // nuovo veicolo System.out.println(b.carPasses("SA00001", 1200)); // 4km in 120 sec (2 minuti), quindi 120km/h System.out.println(c.carPasses("NA12345", 1320)); // 4km in 180 sec (3 minuti), quindi 80km/h System.out.println(c.carPasses("SA00001", 1380));</pre> | <pre>-1 9 -1 120 80</pre> |

192. (UML, 2009-4-23)

Nell'ambito di un programma per la progettazione del software, si implementino la classi UML-Class e UMLAggregation, che rappresentano una classe ed una relazione di aggregazione, all'interno di un diagramma delle classi UML. Il costruttore di UMLAggregation accetta le due classi tra le quali vale l'aggregazione, la cardinalità minima e quella massima.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre> UMLClass impianto = new UMLClass("Impianto"); UMLClass apparecchio = new UMLClass("Apparecchio"); UMLClass contatore = new UMLClass("Contatore"); new UMLAggregation(apparecchio, impianto, 1, 1); new UMLAggregation(impianto, apparecchio, 0, UMLAggregation.INFINITY); new UMLAggregation(impianto, contatore, 0, 1); System.out.println(impianto); </pre> | <p>Output dell'esempio d'uso: Classe: Impianto Aggregazioni: Impianto-Apparecchio, cardinalita': 0..infinito Impianto-Contatore, cardinalita': 0..1</p> |
|--|---|

193. (Container, 2009-2-19)

Si implementi la classe Container, che rappresenta un contenitore per liquidi di dimensione fissata. Ad un contenitore, inizialmente vuoto, si può aggiungere acqua con il metodo addWater, che prende come argomento il numero di litri. Il metodo getAmount restituisce la quantità d'acqua presente nel contenitore. Il metodo connect prende come argomento un altro contenitore, e lo collega a questo con un tubo. Dopo il collegamento, la quantità d'acqua nei due contenitori (e in tutti quelli ad essi collegati) sarà la stessa.

| | |
|---|--|
| <p>Esempio d'uso:</p> <pre> Container a = new Container(), b = new Container(), c = new Container(), d = new Container(); a.addWater(12); d.addWater(8); a.connect(b); System.out.println(a.getAmount()+" "+b.getAmount()+" "+ c.getAmount()+" "+d.getAmount()); b.connect(c); System.out.println(a.getAmount()+" "+b.getAmount()+" "+ c.getAmount()+" "+d.getAmount()); c.connect(d); System.out.println(a.getAmount()+" "+b.getAmount()+" "+ c.getAmount()+" "+d.getAmount()); </pre> | <p>Output dell'esempio d'uso: 6.0 6.0 0.0 8.0 4.0 4.0 4.0 8.0 5.0 5.0 5.0 5.0</p> |
|---|--|

194. (CountByType, 2009-11-27)

Implementare il metodo statico countByType che, data una lista di oggetti, stampa a video il numero di oggetti contenuti nella lista, *divisi in base al loro tipo effettivo*.

Attenzione: il metodo deve funzionare con qualunque tipo di lista e di oggetti contenuti.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre> List<Number> l = new LinkedList<Number>(); l.add(new Integer(3)); l.add(new Double(4.0)); l.add(new Float(7.0f)); l.add(new Integer(11)); countByType(l); </pre> | <p>Output dell'esempio d'uso: java.lang.Double : 1 java.lang.Float : 1 java.lang.Integer : 2</p> |
|--|---|

195. (Volo e Passeggero, 2009-1-15)

Si implementino la classe **Volo** e la classe **Passeggero**. Il costruttore della classe **Volo** prende come argomenti l'istante di partenza e l'istante di arrivo del volo (due numeri interi). Il metodo **add** permette di aggiungere un passeggero a questo volo. Se il passeggero che si tenta di inserire è già presente in un volo che si accavalla con questo, il metodo **add** lancia un'eccezione.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre> Volo v1 = new Volo(1000, 2000); Volo v2 = new Volo(1500, 3500); Volo v3 = new Volo(3000, 5000); Passeggero mario = new Passeggero("Mario"); Passeggero luigi = new Passeggero("Luigi"); v1.add(mario); v1.add(luigi); v3.add(mario); <i>// La seguente istruzione provoca l'eccezione</i> v2.add(mario); </pre> | <p>Output dell'esempio d'uso: Exception in thread "main"...</p> |
|--|---|

196. (PostIt, 2008-9-8)

Un oggetto di tipo **PostIt** rappresenta un breve messaggio incollato (cioè, collegato) ad un oggetto. Il costruttore permette di specificare il messaggio e l'oggetto al quale incollarlo. Il metodo statico **getMessages** prende come argomento un oggetto e restituisce l'elenco dei **PostIt** collegati a quell'oggetto, sotto forma di una lista, oppure **null** se non c'è nessun **PostIt** collegato.

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre> Object frigorifero = new Object(); Object libro = new Object(); new PostIt(frigorifero, "comprare_il_latte"); new PostIt(libro, "Bello !! "); new PostIt(libro, " restituire_a_Carlo"); List<PostIt> pl = PostIt.getMessages(libro); for (PostIt p: pl) System.out.println(p); </pre> | <p>Output dell'esempio d'uso: Bello!! restituire a Carlo</p> |
|--|--|

197. (Molecola, 2008-6-19)

Nell'ambito di un programma di chimica, si implementino le classi **Elemento** e **Molecola**. Un elemento è rappresentato solo dalla sua sigla ("O" per ossigeno, etc.). Una molecola è rappresentata dalla sua formula bruta ("H₂O" per acqua, etc.), cioè dal numero di atomi di ciascun elemento presente.

| | |
|---|--|
| <p>Esempio d'uso:</p> <pre> Elemento ossigeno = new Elemento("O"); Elemento idrogeno = new Elemento("H"); Molecola acqua = new Molecola(); acqua.add(idrogeno, 1); acqua.add(ossigeno, 1); acqua.add(idrogeno, 1); System.out.println(acqua); </pre> | <p>Output dell'esempio d'uso: H2 O</p> |
|---|--|

198. (Impianto e Apparecchio, 2008-3-27)

Si implementi una classe **Impianto** che rappresenta un impianto elettrico, e una classe **Apparecchio** che rappresenta un apparecchio elettrico collegabile ad un impianto. Un impianto è caratterizzato dalla sua potenza massima erogata (in Watt). Ciascun apparecchio è caratterizzato dalla sua

potenza assorbita (in Watt). Per quanto riguarda la classe `Impianto`, il metodo `collega` collega un apparecchio a questo impianto, mentre il metodo `potenza` restituisce la potenza attualmente assorbita da tutti gli apparecchi *collegati* all'impianto ed *accesi*.

I metodi `on` e `off` di ciascun apparecchio accendono e spengono, rispettivamente, questo apparecchio. Se, accendendo un apparecchio col metodo `on`, viene superata la potenza dell'impianto a cui è collegato, deve essere lanciata una eccezione.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Apparecchio tv = new Apparecchio(150); Apparecchio radio = new Apparecchio(30); Impianto i = new Impianto(3000); i.collega(tv); i.collega(radio); System.out.println(i.potenza()); tv.on(); System.out.println(i.potenza()); radio.on(); System.out.println(i.potenza());</pre> | <pre>0 150 180</pre> |

199. (**BoolExpr**, 2008-2-25)

La classe (o interfaccia) `BoolExpr` rappresenta un'espressione dell'algebra booleana (ovvero un circuito combinatorio). Il tipo più semplice di espressione è una semplice variabile, rappresentata dalla classe `BoolVar`, sottotipo di `BoolExpr`. Espressioni più complesse si ottengono usando gli operatori di tipo *and*, *or* e *not*, corrispondenti ad altrettante classi sottotipo di `BoolExpr`. Tutte le espressioni hanno un metodo `eval` che, dato il valore assegnato alle variabili, restituisce il valore dell'espressione. Si consideri *attentamente* il seguente caso d'uso.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>public static void main(String args[]) { BoolVar x = new BoolVar("x"); BoolVar y = new BoolVar("y"); BoolExpr notx = new BoolNot(x); BoolExpr ximpliesy = new BoolOr(notx, y); Map<BoolVar, Boolean> m = new HashMap<BoolVar, Boolean>() ; m.put(x, true); m.put(y, true); System.out.println(x.eval(m)); System.out.println(ximpliesy.eval(m)); m.put(y, false); System.out.println(ximpliesy.eval(m)); }</pre> | <pre>true true false</pre> |

200. (**Recipe**, 2008-1-30)

Si implementi una classe `Recipe` che rappresenta una ricetta. Il costruttore accetta il nome della ricetta. Il metodo `setDescr` imposta la descrizione della ricetta. Il metodo `addIngr` aggiunge un ingrediente alla ricetta, prendendo come primo argomento la quantità (anche frazionaria) dell'ingrediente, per una persona, e come secondo argomento una stringa che contiene l'unità di misura e il nome dell'ingrediente. Se un ingrediente è difficilmente misurabile, si imposterà la sua quantità a zero, e verrà visualizzato come "q.b." ("quanto basta"). Il metodo `toString` prende come argomento il numero di coperti *n* e restituisce una stringa che rappresenta la ricetta, in cui le quantità degli ingredienti sono state moltiplicate per *n*.

| | |
|--|---|
| Esempio d'uso: | Output dell'esempio d'uso: Spaghetti aglio e olio Ingredienti per 4 persone: 400 grammi di spaghetti 8 cucchiai d'olio d'oliva 4 spicchi d'aglio q.b. sale Preparazione: Mischiare tutti gli ingredienti e servire. |
| <pre>Recipe r = new Recipe("Spaghetti_aglio_e_olio"); r.addIngr(100, "grammi_di_spaghetti"); r.addIngr(2, "cucchiai_d'olio_d'oliva"); r.addIngr(1, "spicchi_d'aglio"); r.addIngr(0, "sale"); r.setDescr("Mischiare_tutti_gli_ingredienti_e_servire."); ; System.out.println(r.toString(4));</pre> | |

201. (FunnyOrder, 2007-9-17)

Determinare l'output del seguente programma e descrivere brevemente l'ordinamento dei numeri interi definito dalla classe FunnyOrder.

```
public class FunnyOrder implements Comparable<FunnyOrder> {
    private int val;
    public FunnyOrder(int n) { val = n; }
    public int compareTo(FunnyOrder x) {
        if (val%2 == 0 && x.val%2 != 0) return -1;
        if (val%2 != 0 && x.val%2 == 0) return 1;
        if (val < x.val) return -1;
        if (val > x.val) return 1;
        return 0;
    }
    public static void main(String[] args) {
        List<FunnyOrder> l = new LinkedList<FunnyOrder>();
        l.add(new FunnyOrder(16));
        l.add(new FunnyOrder(3));
        l.add(new FunnyOrder(4));
        l.add(new FunnyOrder(10));
        l.add(new FunnyOrder(2));
        Collections.sort(l);
        for (FunnyOrder f: l)
            System.out.println(f.val + " ");
    }
}
```

202. (2007-6-29)

Individuare gli errori di compilazione nella seguente classe. Commentare brevemente ciascun errore e fornire una possibile correzione.

```
1 public class Errors {
2     private static int num = 7;
3     private Integer z = 8;
4     Map<Integer, Errors> m = new Map<Integer, Errors>();
5
6     public Errors() { }
7
8     private static class A {
9         private A() { num += z; }
10    }
11    private void f() {
12        m.put(7, new Errors() { public int g() { return 0; } });
13    }
14
15    public static final A a = new A();
16 }
```

203. (Highway, 2007-6-29)

Implementare una classe `Highway`, che rappresenti un'autostrada a senso unico. Il costruttore accetta la lunghezza dell'autostrada in chilometri. Il metodo `insertCar` prende un intero x come argomento ed aggiunge un'automobile al chilometro x . L'automobile inserita percorrerà l'autostrada alla velocità di un chilometro al minuto, (60 km/h) fino alla fine della stessa. Il metodo `nCars` prende un intero x e restituisce il numero di automobili presenti al chilometro x . Il metodo `progress` simula il passaggio di 1 minuto di tempo (cioè fa avanzare tutte le automobili di un chilometro).

Si supponga che thread multipli possano accedere allo stesso oggetto `Highway`.

Dei 25 punti, 8 sono riservati a coloro che implementeranno `progress` in tempo indipendente dal numero di automobili presenti sull'autostrada.

| | |
|--|---------|
| Esempio d'uso: | Output: |
| <pre>Highway h = new Highway(10); h.insertCar(3); h.insertCar(3); h.insertCar(5); System.out.println(h.nCars(4)); h.progress(); System.out.println(h.nCars(4));</pre> | 0 |
| | 2 |

204. (Polinomio bis, 2007-2-7)

Si consideri la seguente classe `Pair`.

```
public class Pair<T, U>
{
    public Pair(T first, U second) { this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public U getSecond() { return second; }

    private T first;
    private U second;
}
```

Un *polinomio* è una espressione algebrica del tipo $a_0 + a_1x + \dots + a_nx^n$. Si implementi una classe `Polynomial`, dotata di un costruttore che accetta un array contenente i coefficienti $a_0 \dots a_n$. Deve essere possibile iterare sulle coppie (esponente, coefficiente) in cui il coefficiente è diverso da zero. Cioè, `Polynomial` deve implementare `Iterable<Pair<Integer, Double>>`. Infine, il metodo `toString` deve produrre una stringa simile a quella mostrata nel seguente caso d'uso.

| | |
|---|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>double a1[] = {1, 2, 0, 3}; double a2[] = {0, 2}; Polynomial p1 = new Polynomial(a1); Polynomial p2 = new Polynomial(a2); System.out.println(p1); System.out.println(p2); for (Pair<Integer, Double> p: p1) System.out.println(p.getFirst() + " : " + p.getSecond());</pre> | <pre>1.0 + 2.0 x^1 + 3.0 x^3 2.0 x^1 0 : 1.0 1 : 2.0 3 : 3.0</pre> |

205. (Inventory, 2007-2-23)

Definire una classe parametrica `Inventory<T>` che rappresenta un inventario di oggetti di tipo `T`. Il costruttore senza argomenti crea un inventario vuoto. Il metodo `add` aggiunge un oggetto di tipo `T` all'inventario. Il metodo `count` prende come argomento un oggetto di tipo `T` e restituisce il numero di oggetti uguali all'argomento presenti nell'inventario. Infine, il metodo `getMostCommon` restituisce l'oggetto di cui è presente il maggior numero di esemplari.

| | |
|--|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Inventory<Integer> a = new Inventory<Integer>(); Inventory<String> b = new Inventory<String>(); a.add(7); a.add(6); a.add(7); a.add(3); b.add("ciao"); b.add("allora?"); b.add("ciao_ciao"); b.add("allora? "); System.out.println(a.count(2)); System.out.println(a.count(3)); System.out.println(a.getMostCommon()); System.out.println(b.getMostCommon());</pre> | <pre>0 1 7 allora?</pre> |

206. (Insieme di polinomi, 2007-1-12)

Con riferimento all'esercizio 142, implementare una classe `PolynomialSet`, che rappresenta un insieme di `Polynomial`. La classe deve offrire almeno i seguenti metodi: `add` accetta un `Polynomial` e lo aggiunge all'insieme; `maxDegree` restituisce il massimo grado dei polinomi dell'insieme; `iterator` restituisce un iteratore sui polinomi dell'insieme. Aggiunere all'insieme un polinomio con gli stessi coefficienti di uno che è già presente non ha alcun effetto sull'insieme.

Dire se nella vostra implementazione è necessario modificare la classe `Polynomial`, e perché.

Dei 25 punti, 7 sono riservati a coloro che forniranno una soluzione in cui `maxDegree` richiede tempo costante (cioè, un tempo indipendente dal numero di polinomi presenti nell'insieme).

207. (Spartito, 2006-7-17)

Nel contesto di un software per la composizione musicale, si implementi una classe `Nota`, e una classe `Spartito`. Ciascuna nota ha un nome e una durata. La durata può essere soltanto di 1, 2, oppure 4 unità di tempo (semiminima, minima oppure semibreve). Uno spartito è una sequenza di note, tale che più note possono cominciare (o terminare) nello stesso istante. Il metodo `add` della classe `Spartito` prende come argomento una nota ed un istante di tempo t , ed aggiunge la nota allo spartito, a partire dal tempo t . Quando si itera su uno spartito, ad ogni chiamata a `next` viene restituito l'insieme di note presenti nell'unità di tempo corrente.

Implementare tutti i metodi necessari a rispettare il seguente caso d'uso.

```
public static void main(String[] x) {
    Spartito fuga = new Spartito();
    fuga.add(new Nota("Do", 4), 0);
    fuga.add(new Nota("Mi", 1), 0);
    fuga.add(new Nota("Mib", 2), 1);
    fuga.add(new Nota("Sol", 2), 2);

    for (Set<Nota> accordo : fuga)
        System.out.println(accordo);
}
```

Output del codice precedente:

```
Do, Mi
Do, Mib
Do, Mib, Sol
Do, Sol
```

208. (2006-7-17)

Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma.

```
1 class Test {
2     Collection<?> c;
3
4     public int f(Collection<? extends Number> c) {
```

```

5     return c.size();
6 }
7
8 public void g(Set<? extends Number> c) {
9     this.c = c;
10}
11
12 private <T extends Number> T myAdd(T x) {
13     c.add(x);
14     return x;
15}
16
17 public static void main(String args[]) {
18     Test t = new Test();
19
20     t.f(new LinkedList<Integer>());
21     t.g(new ArrayList<Integer>());
22     t.myAdd(new Double(2.0));
23 }
24 }
```

209. (Publication, 2006-6-26)

Nel contesto di un software per biblioteche, si implementi una classe **Publication**, che rappresenta una pubblicazione. Ciascuna pubblicazione ha un titolo e una data di uscita. Implementare le sottoclassi **Book** e **Magazine**. Un libro (book) ha anche un codice ISBN (numero intero di 13 cifre), mentre una rivista (magazine) ha un numero progressivo. Inoltre, una pubblicazione può fare riferimento ad altre pubblicazioni tramite riferimenti bibliografici.

Implementare tutti i metodi necessari a rispettare il seguente caso d'uso.

```

public static void main(String[] x) {
    Publication libro = new Book("The_Art_of_Unix_Programming", new Date(1990, 3, 24),
        123456);
    Publication rivista = new Magazine("Theoretical_Computer_Science", new Date(1985, 4, 13), 74);
    rivista.addReference(libro);

    for (Publication p : rivista.references())
        System.out.println(p);

    libro.addReference(libro);
}
```

Output del codice precedente:

The Art of Unix Programming, ISBN: 123456

```

Exception in thread "main" java.lang.RuntimeException
    at Publication.addReference(PublicationTest.java:13)
    at PublicationTest.main(PublicationTest.java:59)
```

210. (DoubleQueue, 2006-6-26)

Implementare la classe **DoubleQueue**, che rappresenta due code con carico bilanciato. Quando viene aggiunto un nuovo elemento alla **DoubleQueue**, l'elemento viene aggiunto alla coda più scarica, cioè a quella che contiene meno elementi.

```

DoubleQueue<Integer> q = new DoubleQueue<Integer>();
q.add(3);
q.add(5);
q.add(7);

System.out.println("Contenuto della prima coda:");
while (!q.isEmpty1())
```

```
System.out.println(q.remove1());  
  
System.out.println("Contenuto della seconda coda:");  
while (!q.isEmpty2())  
    System.out.println(q.remove2());
```

Output del codice precedente:

```
Contenuto della prima coda:  
3  
7  
Contenuto della seconda coda:  
5
```

211. (2006-6-26)

Individuare e descrivere sinteticamente gli errori nel seguente programma.

```
1 class Test {  
2     public static void f(List<? extends Number> l) {  
3         l.add(new Integer(3));  
4     }  
5     public static <T> T myGet(Map<T, ? extends T> m, T x) {  
6         return m.get(x);  
7     }  
8     public static void main(String args[]) {  
9         f(new LinkedList<Integer>());  
10        f(new ArrayList<Boolean>());  
11        f(new List<Double>());  
12        Object o = myGet(new HashMap<Number, Integer>(), new Integer(7));  
13    }  
14 }  
15 }
```

5 Scelta della firma

212. (keysWithValue, 2022-2-24)

Implementare il metodo `keysWithValue`, che accetta una mappa con valori interi e restituisce l'insieme delle chiavi associate al valore massimo.

Inoltre, valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `Set<?> keysWithValue(Map<?, Integer> map)`
- b) `Set<Object> keysWithValue(Map<Object, Integer> map)`
- c) `<S> Set<S> keysWithValue(Map<S, ? super Integer> map)`
- d) `<S> Set<S> keysWithValue(Map<S, ? extends Integer> map)`
- e) `<S> Set<? super S> keysWithValue(Map<? extends S, Integer> map)`
- f) `<S> Set<S> keysWithValue(Map<S, Integer> map)`

213. (combine, 2022-1-26)

Implementare il metodo `combine`, che accetta due comparatori e li combina *lessicograficamente*, ovvero restituisce un comparatore che, dati due oggetti, restituisce il valore del primo comparatore, se diverso da zero; altrimenti, restituisce il valore del secondo comparatore.

Inoltre, valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `<T> Comparator<T> combine(Comparator<T> a, Comparator<T> b)`
- b) `<T> Comparator<T> combine(Comparator<T> a, Comparator<?> b)`
- c) `<S, T extends S, U extends S> Comparator<S> combine(Comparator<T> a, Comparator<U> b)`
- d) `<T> Comparator<T> combine(Comparator<? super T> a, Comparator<? super T> b)`
- e) `<T> Comparator<T> combine(Comparator<Object> a, Comparator<Object> b)`
- f) `<T> Comparator<? extends T> combine(Comparator<? super T> a, Comparator<? super T> b)`

214. (countOccurrences, 2021-9-24)

Implementare il metodo `countOccurrences`, che accetta una collezione e restituisce una mappa che, ad ogni oggetto della collezione, associa il numero di ripetizioni presenti.

Inoltre, valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `<K> Map<? extends K, Integer> countOccurrences(Collection<Object> c)`
- b) `<K> Map<? extends K, Integer> countOccurrences(Collection<? super K> c)`
- c) `Map<Object, Integer> countOccurrences(Collection<Object> c)`
- d) `<S, T extends S> Map<S, Integer> countOccurrences(Collection<T> c)`
- e) `<K> Map<? super K, Integer> countOccurrences(Collection<? extends K> c)`
- f) `Map<Object, Integer> countOccurrences(Collection<?> c)`

215. (overridingMap, 2021-7-26)

Il metodo `overridingMap`, accetta due mappe e restituisce una nuova mappa che ha le stesse chiavi della prima e i seguenti valori: se una chiave compare solo nella prima mappa, il suo valore sarà quello associato nella prima mappa; se una chiave compare anche nella seconda mappa, il suo valore sarà quello associato nella seconda mappa.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K,V> Map<?,?> overridingMap(Map<K,V> map1, Map<K,V> map2)
- b) <K> Map<K,?> overridingMap(Map<K,?> map1, Map<K,?> map2)
- c) <K,V> Map<K,V> overridingMap(Map<K,V> map1, Map<? extends K,?> map2)
- d) <K,V> Map<K,V> overridingMap(Map<K,? extends V> map1, Map<?,? extends V> map2)
- e) <K,V> Map<? extends K,V> overridingMap(Map<? super K,V> map1, Map<? super K,V> map2)
- f) <K> Map<K,Object> overridingMap(Map<K,Object> map1, Map<?,Object> map2)

216. (countInBetween, 2021-10-26)

Implementare il metodo statico `countInBetween`, che accetta un array, un comparatore e due oggetti *a* e *b*, e restituisce il numero di oggetti dell'array che, secondo il comparatore, sono maggiori di *a* e minori di *b*.

Porre particolare attenzione alla scelta dell'intestazione.

217. (keysWithHighestValue, 2020-2-27)

Implementare il metodo `keysWithHighestValue`, che accetta una mappa e un comparatore, e restituisce l'insieme delle chiavi che hanno il massimo valore associato, secondo il comparatore.

Ad esempio, se una `Map<String, Integer>` contiene le coppie `a:5, b:3, c:0, d:5`, e il comparatore rispecchia l'ordine naturale tra interi, il metodo deve restituire l'insieme `{a, d}`.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K> Set<K> keysWithHighestValue(Map<K, Object> map, Comparator<Object> c)
- b) <T> Set<T> keysWithHighestValue(Map<T, T> map, Comparator<T> c)
- c) <K,V> Set<K> keysWithHighestValue(Map<K, V> map, Comparator<?> c)
- d) <V> Set<Object> keysWithHighestValue(Map<?,? extends V> map, Comparator<? super V> c)
- e) <K,V> Set<K> keysWithHighestValue(Map<? super K,? extends V> map, Comparator<? super V> c)

218. (disjoin, 2019-9-20)

Implementare il metodo `disjoin`, che accetta due collezioni e rimuove da entrambe tutti gli oggetti che hanno in comune. Inoltre, restituisce l'insieme degli oggetti rimossi, senza ripetizioni.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <S> Set<S> disjoin(Collection<S> a, Collection<S> b)
- b) <S> Set<S> disjoin(Collection<? extends S> a, Collection<?> b)
- c) <S,T> Set<? super S> disjoin(Collection<S> a, Collection<T> b)
- d) Set<Object> disjoin(List<?> a, List<?> b)
- e) <S> Set<S> disjoin(Collection<? super S> a, Collection<? super S> b)

219. (Minimum enum, 2019-7-23)

Implementare il metodo `min`, che accetta due elementi di classi enumerate. Se i due oggetti appartengono alla stessa classe enumerata, il metodo restituisce quello con l'ordinale minore; altrimenti, restituisce `null`.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. In caso di non completezza, presentare un controesempio.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `Enum<?> min(Enum<?> a, Enum<?> b)`
- b) `<T> Enum<T> min(Enum<T> a, Enum<T> b)`
- c) `<S extends Enum<S>> S min(S a, S b)`
- d) `<S extends Enum<S>> S min(S a, Object b)`
- e) `<T> Enum<T> min(Enum<T> a, Enum<?> b)`

220. (keysWithValue, 2019-6-24)

Il metodo `keysWithValue` accetta una mappa, un valore e una lista, e inserisce nella lista tutte le chiavi della mappa che hanno quel valore associato.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. In caso di non completezza, indicare un controesempio.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `<K,V> void keysWithValue(Map<K,V> m, V value, List<K> out)`
- b) `<K> void keysWithValue(Map<K,> m, Object value, List<Object> out)`
- c) `<K,V> void keysWithValue(Map<? extends K,V> m, V value, List<K> out)`
- d) `<K,V> void keysWithValue(Map<? extends K,V> m, V value, List<? extends K> out)`
- e) `<V> void keysWithValue(Map<?,V> m, V value, List<?> out)`
- f) `<K> void keysWithValue(Map<K,> m, Object value, LinkedList<? super K> out)`

221. (interleave2, 2019-10-9)

Implementare un metodo statico `interleave` che prende come argomento tre liste *A*, *B* e *C*. Senza modificare *A* e *B*, il metodo aggiunge tutti gli elementi di *A* e di *B* a *C*, in modo alternato (prima un elemento di *A*, poi uno di *B*, poi un altro elemento di *A*, e così via).

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità.

Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `<S> void interleave(List<S> a, List<S> b, List<S> c)`
- b) `<S, T extends S> void interleave(List<T> a, List<T> b, List<S> c)`
- c) `void interleave(List<?> a, List<?> b, List<Object> c)`
- d) `<S> void interleave(List<?> a, List<?> b, List<S> c)`
- e) `<S> void interleave(List<S> a, List<S> b, List<? super S> c)`

222. (findPrevious, 2019-1-23)

Implementare il metodo statico `findPrevious`, che accetta un insieme, un comparatore e un oggetto *x*, e restituisce il più grande oggetto dell'insieme che è minore di *x* (secondo il comparatore). Se tale oggetto non esiste (perché tutti gli elementi dell'insieme sono maggiori o uguali a *x*), il metodo restituisce `null`.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

5 Scelta della firma

- a) $\langle T \rangle T \text{ findPrevious}(\text{Set} < ? \text{ extends } T > \text{ set}, \text{ Comparator} < ? > \text{ comp}, T x)$
- b) $\langle S, T \text{ extends } S \rangle T \text{ findPrevious}(\text{Set} < T > \text{ set}, \text{ Comparator} < S > \text{ comp}, T x)$
- c) $\langle S, T \text{ extends } S \rangle S \text{ findPrevious}(\text{Set} < S > \text{ set}, \text{ Comparator} < T > \text{ comp}, S x)$
- d) $\langle T \rangle T \text{ findPrevious}(\text{Set} < T > \text{ set}, \text{ Comparator} < ? \text{ super } T > \text{ comp}, T x)$
- e) $\langle T \rangle T \text{ findPrevious}(\text{Set} < T > \text{ set}, \text{ Comparator} < T > \text{ comp}, T x)$
- f) $\langle T \rangle T \text{ findPrevious}(\text{Set} < ? \text{ super } T > \text{ set}, \text{ Comparator} < T > \text{ comp}, T x)$

223. (makeMap, 2018-6-20)

Il metodo statico `makeMap` accetta una lista di chiavi e una lista di valori (di pari lunghezza), e restituisce una mappa ottenuta accoppiando ciascun elemento della prima lista al corrispondente elemento della seconda lista.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) $\langle K, V \rangle \text{ Map} < ? \text{ extends } K, ? \text{ extends } V > \text{ makeMap}(\text{List} < K > \text{ keys}, \text{List} < V > \text{ vals})$
- b) $\langle K, V \rangle \text{ Map} < ? \text{ extends } K, ? > \text{ makeMap}(\text{List} < K > \text{ keys}, \text{List} < ? > \text{ vals})$
- c) $\langle K, V \rangle \text{ Map} < K, V > \text{ makeMap}(\text{List} < K > \text{ keys}, \text{List} < ? > \text{ vals})$
- d) $\langle T \rangle \text{ Map} < T, T > \text{ makeMap}(\text{List} < ? \text{ extends } T > \text{ keys}, \text{List} < ? \text{ extends } T > \text{ vals})$
- e) $\langle K \rangle \text{ Map} < K, ? > \text{ makeMap}(\text{List} < K > \text{ keys}, \text{List} < \text{Object} > \text{ vals})$
- f) $\langle K, V \text{ extends } K \rangle \text{ Map} < K, V > \text{ makeMap}(\text{List} < K > \text{ keys}, \text{List} < V > \text{ vals})$

224. (cartesianProduct, 2018-2-22)

Data una classe `Pair<S,T>`, il metodo statico `cartesianProduct` accetta due insiemi e restituisce il loro prodotto cartesiano.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) $\langle S, T \rangle \text{ Set} < \text{Pair} < S, T > > \text{ cartesianProduct}(\text{Set} < S > s, \text{Set} < T > t)$
- b) $\langle S, T \rangle \text{ Set} < \text{Pair} < S, T > > \text{ cartesianProduct}(\text{Set} < ? > s, \text{Set} < ? > t)$
- c) $\text{Set} < \text{Pair} < ?, ? > > \text{ cartesianProduct}(\text{Set} < \text{Object} > s, \text{Set} < \text{Object} > t)$
- d) $\langle S, T \rangle \text{ Set} < ? > \text{ cartesianProduct}(\text{Set} < S > s, \text{Set} < T > t)$
- e) $\langle S \rangle \text{ Set} < \text{Pair} < S, S > > \text{ cartesianProduct}(\text{Set} < S > s, \text{Set} < S > t)$
- f) $\langle S, T \text{ extends } S \rangle \text{ Set} < \text{Pair} < S, T > > \text{ cartesianProduct}(\text{Set} < S > s, \text{Set} < T > t)$

225. (greatestLowerBound, 2018-10-18)

Implementare il metodo `gLb` (per *greatestLowerBound*), che accetta due insiemi A e B, e un comparatore, e restituisce il più grande elemento di A che è più piccolo di tutti gli elementi di B. Se un tale elemento non esiste, il metodo restituisce `null`.

Ad esempio, se $A = \{5, 20, 30\}$ e $B = \{25, 26, 30\}$, il metodo deve restituire 20.

Inoltre, valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) $\langle T \rangle T \text{ gLB}(\text{Set} < ? \text{ extends } T > a, \text{Set} < ? \text{ extends } T > b, \text{ Comparator} < T > c)$
- b) $\langle S, T \rangle \text{ Object} \text{ gLB}(\text{Set} < S > a, \text{Set} < T > b, \text{ Comparator} < \text{Object} > c)$
- c) $\langle T \rangle T \text{ gLB}(\text{Set} < T > a, \text{Set} < T > b, \text{ Comparator} < ? \text{ super } T > c)$
- d) $\langle S, T \text{ extends } S \rangle S \text{ gLB}(\text{Set} < S > a, \text{Set} < T > b, \text{ Comparator} < S > c)$
- e) $\langle T \rangle T \text{ gLB}(\text{Set} < ? \text{ super } T > a, \text{Set} < ? \text{ super } T > b, \text{ Comparator} < T > c)$

226. (**isIncreasing**, 2018-1-24)

Il metodo statico `isIncreasing` accetta una mappa e un comparatore, e restituisce vero se e solo se ciascuna chiave è minore o uguale del valore ad essa associato.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K,V> **boolean** `isIncreasing(Map<K,V> m, Comparator<K> c)`
- b) <K,V> **boolean** `isIncreasing(Map<K,V> m, Comparator<? super K> c)`
- c) <K,V extends K> **boolean** `isIncreasing(Map<K,V> m, Comparator<? super K> c)`
- d) <T> **boolean** `isIncreasing(Map<T,T> m, Comparator<T> c)`
- e) <T> **boolean** `isIncreasing(Map<T,T> m, Comparator<? extends T> c)`
- f) <T> **boolean** `isIncreasing(Map<? extends T, ? extends T> m, Comparator<T> c)`
- g) **boolean** `isIncreasing(Map<?,?> m, Comparator<?> c)`

227. (**commonKeys**, 2017-7-20)

Implementare un metodo statico `commonKeys`, che accetta due mappe, e restituisce l'insieme degli oggetti che compaiono come chiavi in entrambe le mappe.

Inoltre, valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <T> `Set<T> commonKeys(Map<T,?> m1, Map<T,?> m2)`
- b) <T,V1,V2> `Set<T> commonKeys(Map<T,V1> m1, Map<T,V2> m2)`
- c) `Set<Object> commonKeys(Map<?,?> m1, Map<?,?> m2)`
- d) <T> `Set<? extends T> commonKeys(Map<? extends T,?> m1, Map<? extends T,?> m2)`
- e) <T> `Set<?> commonKeys(Map<T,?> m1, Map<?,?> m2)`

228. (**findNext**, 2017-6-21)

Il metodo statico `findNext` accetta un insieme, un comparatore e un oggetto *x*, e restituisce il più piccolo oggetto dell'insieme che è maggiore di *x* (secondo il comparatore).

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <T> `T findNext(Set<? extends T> set, Comparator<?> comp, T x)`
- b) <S,T extends S> `T findNext(Set<T> set, Comparator<S> comp, T x)`
- c) <S,T extends S> `S findNext(Set<S> set, Comparator<T> comp, S x)`
- d) <T> `T findNext(Set<T> set, Comparator<? super T> comp, T x)`
- e) <T> `T findNext(Set<T> set, Comparator<T> comp, Object x)`

229. (**arePermutations**, 2016-6-22)

Il metodo statico `arePermutations`, accetta due liste e controlla se contengono gli stessi elementi (secondo `equals`), anche in ordine diverso.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Scegliere l'intestazione migliore oppure proporne un'altra. Infine, implementare il metodo usando l'intestazione prescelta.

- a) **boolean** `arePermutations(List<?> a, List<?> b)`
- b) <S,T> **boolean** `arePermutations(List<S> a, List<T> b)`
- c) <S> **boolean** `arePermutations(List<S> a, List<S> b)`
- d) <S> **boolean** `arePermutations(List<? extends S> a, List<? extends S> b)`

5 Scelta della firma

- e) **boolean** arePermutations(List<Object> a, List<Object> b)
- f) <S, T extends S> **boolean** arePermutations(List<? extends S> a, List<? extends T> b)

230. (splitList, 2015-9-21)

Il metodo statico `splitList` spezza una lista `src` in due parti, inserendo in una lista `part1` tutti gli elementi che vengono prima di un dato elemento `x`, e tutti gli altri elementi in una lista chiamata `part2`.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <T> **void** splitList(List<T> src, T x, List<T> part1, List<T> part2)
- b) **void** splitList (List<Object> src, Object x, List<?> part1, List<?> part2)
- c) <S,T> **void** splitList(List<S> src, S x, List<T> part1, List<T> part2)
- d) <T> **void** splitList(List<? extends T> src, T x, List<T> part1, List<T> part2)
- e) <T> **void** splitList(List<T> src, Object x, List<? super T> part1, List<? super T> part2)

231. (listIntersection, 2015-6-24)

Implementare il metodo statico `listIntersection`, che accetta una lista e un insieme, e restituisce una nuova lista, che contiene gli elementi della lista che appartengono anche all'insieme, nello stesso ordine in cui appaiono nella prima lista.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) List<?> listIntersection (List<?> l, Set<?> s)
- b) List<Object> listIntersection(List<Object> l, Set<?> s)
- c) <T> List<T> listIntersection(List<T> l, Set<? extends T> s)
- d) <T> List<T> listIntersection(List<T> l, Set<?> s)
- e) <S,T> List<T> listIntersection(List<T> l, Set<S> s)

232. (reverseList, 2015-2-5)

Il metodo `reverseList` accetta una lista e restituisce una nuova lista, che contiene gli stessi elementi della prima, ma in ordine inverso.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) List<?> reverseList(List<?> l)
- b) <T> List<? extends T> reverseList(List<? super T> l)
- c) <T extends List<?>> T reverseList(T l)
- d) <T> List<T> reverseList(List<T> l)
- e) List<Object> reverseList(List<Object> l)

233. (difference, 2015-1-20)

Il metodo `difference` accetta due insiemi `a` e `b` e restituisce un nuovo insieme, che contiene gli elementi che appartengono ad `a` ma non a `b` (cioè, la differenza insiemistica tra `a` e `b`).

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) Set<?> difference(Set<?> a, Set<?> b)
- b) Set<Object> difference(Set<?> a, Set<?> b)

- c) Set<Object> difference(Set<String> a, Set<String> b)
- d) <T> Set<T> difference(Set<T> a, Set<?> b)
- e) <T> Set<T> difference(Set<? extends T> a, Set<? extends T> b)
- f) <T> Set<T> difference(Set<T> a, Set<? extends T> b)

234. (subMap, 2014-7-3)

Implementare il metodo **subMap** che accetta una mappa e una collezione e restituisce una nuova mappa ottenuta restringendo la prima alle sole chiavi che compaiono nella collezione. Il metodo non modifica i suoi argomenti.

Valutare le seguenti intestazioni per il metodo **subMap**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K> Map<K,?> subMap(Map<K,?> m, Collection<K> c)
- b) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<?> c)
- c) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<? super K> c)
- d) <K,V> Map<K,V> subMap(Map<K,V> m, Collection<? extends K> c)
- e) <K,V> Map<K,V> subMap(Map<K,V> m, Set<K> c)
- f) <K,V,K2 extends K> Map<K,V> subMap(Map<K,V> m, Collection<K2> c)

235. (inverseMap, 2014-7-28)

Implementare il metodo **inverseMap** che accetta una mappa **m** e ne restituisce una nuova, ottenuta invertendo le chiavi con i valori. Se **m** contiene valori duplicati, il metodo lancia un'eccezione. Il metodo non modifica la mappa **m**.

Valutare le seguenti intestazioni per il metodo **inverseMap**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) <K,V> Map<V,K> inverseMap(Map<?,?> m)
- b) Map<?,?> inverseMap(Map<?,?> m)
- c) <K,V> Map<K,V> inverseMap(Map<V,K> m)
- d) <K,V> Map<K,V> inverseMap(Map<? extends V, ? super K> m)
- e) <K,V> Map<K,V> inverseMap(Map<K,V> m)
- f) <K,V> Map<K,V> inverseMap(Map<? extends V, ? extends K> m)

236. (extractPos, 2014-3-5)

Il metodo **extractPos** accetta una lista ed un numero intero **n** e restituisce l'ennesimo elemento della lista.

Valutare ciascuna delle seguenti intestazioni per il metodo **extractPos**, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra, motivando brevemente la propria scelta.

- a) Object extractPos(Collection<?> l, int n)
- b) <T> T extractPos(List<T> l, int n)
- c) <T> T extractPos(List<? extends T> l, int n)
- d) Object extractPos(List<?> l, int n)
- e) <T> T extractPos(LinkedList<T> l, int n)
- f) <S,T> S extractPos(List<T> l, int n)

5 Scelta della firma

237. (product, 2014-11-28)

Il metodo `product` accetta due insiemi e restituisce il loro prodotto Cartesiano.

Valutare ciascuna delle seguenti intestazioni in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `Set<Pair<?,?>> product(Set<?> a, Set<?> b)`
- b) `<S,T> Set<Pair<S,T>> product(Set<S> a, Set<T> b)`
- c) `Set<Pair<Object, Object>> product(Set<Object> a, Set<Object> b)`

238. (isMax, 2014-1-31)

Il metodo `isMax` accetta un oggetto `x`, un comparatore ed un insieme di oggetti, e restituisce `true` se, in base al comparatore, `x` è maggiore o uguale di tutti gli oggetti contenuti nell'insieme. Altrimenti, il metodo restituisce `false`.

Valutare ciascuna delle seguenti intestazioni per il metodo `isMax`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra, motivando brevemente la propria scelta.

- a) `boolean isMax(Object x, Comparator<Object> c, Set<Object> s)`
- b) `<T> boolean isMax(T x, Comparator<T> c, Set<T> s)`
- c) `<T> boolean isMax(T x, Comparator<? super T> c, Set<T> s)`
- d) `<T> boolean isMax(T x, Comparator<? extends T> c, Set<? super T> s)`
- e) `<T> boolean isMax(T x, Comparator<? super T> c, Set<? super T> s)`
- f) `<S,T extends S> boolean isMax(T x, Comparator<? super S> c, Set<S> s)`

239. (composeMaps, 2013-9-25)

Il metodo `composeMaps` accetta due mappe `a` e `b`, e restituisce una nuova mappa `c` così definita: le chiavi di `c` sono le stesse di `a`; il valore associato in `c` ad una chiave `x` è pari al valore associato nella mappa `b` alla chiave `a(x)`.

Nota: Se consideriamo le mappe come funzioni matematiche, la mappa c è definita come $c(x) = b(a(x))$, cioè come composizione di a e b.

Valutare ciascuna delle seguenti intestazioni per il metodo `composeMaps`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, specificità del tipo di ritorno e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `<S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<T, U> b)`
- b) `<S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<? extends T, U> b)`
- c) `<S, T, U> Map<S,U> composeMaps(Map<S, T> a, Map<? super T, U> b)`
- d) `<S, U> Map<S,U> composeMaps(Map<S, ?> a, Map<?, U> b)`
- e) `<S, U> Map<S,U> composeMaps(Map<S, Object> a, Map<Object, U> b)`

240. (isSorted, 2013-7-9)

Implementare il metodo `isSorted` che accetta una lista e un comparatore, e restituisce `true` se la lista risulta già ordinata in senso non-decrescente rispetto a quel comparatore, e `false` altrimenti.

Valutare ciascuna delle seguenti intestazioni per il metodo `isSorted`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `boolean isSorted(List<?> x, Comparator<Object> c)`
- b) `<S> boolean isSorted(List<? extends S> x, Comparator<? super S> c)`
- c) `<S> boolean isSorted(List<S> x, Comparator<S> c)`
- d) `boolean isSorted(List<Object> x, Comparator<Object> c)`

- e) <S, T> **boolean** `isSorted(List<S> x, Comparator<T> c)`
- f) <S, T extends S> **boolean** `isSorted(List<T> x, Comparator<S> c)`

241. (**Concat, 2013-6-25**)

Implementare il metodo `concat` che accetta due iteratori *a* e *b* e restituisce un altro iteratore che restituisce prima tutti gli elementi restituiti da *a* e poi tutti quelli di *b*.

Valutare le seguenti intestazioni per il metodo `concat`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie, semplicità e specificità del tipo di ritorno. Infine, scegliere l'intestazione migliore oppure proporne un'altra.

- a) `Iterator<Object> concat(Iterator<Object> a, Iterator<Object> b)`
- b) `Iterator<?> concat(Iterator<?> a, Iterator<?> b)`
- c) <S> `Iterator<S> concat(Iterator<S> a, Iterator<S> b)`
- d) <S> `Iterator<S> concat(Iterator<? extends S> a, Iterator<? extends S> b)`
- e) <S,T> `Iterator<S> concat(Iterator<S> a, Iterator<T> b)`
- f) <S,T extends S> `Iterator<T> concat(Iterator<T> a, Iterator<S> b)`

242. (**agree, 2013-12-16**)

Il metodo `agree` accetta due oggetti `Comparator c1` e `c2` e altri due oggetti `a` e `b`, e restituisce `true` se i due comparatori concordano sull'ordine tra `a` e `b` (cioè, l'esito delle invocazioni `c1.compare(a,b)` e `c2.compare(a,b)` è lo stesso) e `false` altrimenti.

Valutare ciascuna delle seguenti intestazioni per il metodo `agree`, in base ai criteri di funzionalità, completezza, correttezza, fornitura di ulteriori garanzie e semplicità. Infine, scegliere l'intestazione migliore oppure proporne un'altra, motivando brevemente la propria scelta.

- a) <T> **boolean** `agree(Comparator<T> c1, Comparator<T> c2, T a, T b)`
- b) **boolean** `agree(Comparator<Object> c1, Comparator<Object> c2, Object a, Object b)`
- c) <S, T> **boolean** `agree(Comparator<S> c1, Comparator<T> c2, S a, T b)`
- d) <T> **boolean** `agree(Comparator<? extends T> c1, Comparator<? extends T> c2, T a, T b)`
- e) <T> **boolean** `agree(Comparator<? super T> c1, Comparator<? super T> c2, T a, T b)`
- f) <S, T extends S> **boolean** `agree(Comparator<S> c1, Comparator<S> c2, T a, T b)`

6 Trova l'errore

243. (2007-7-20)

Individuare gli errori di *compilazione* nella seguente classe. Commentare brevemente ciascun errore e fornire una possibile correzione.

```
1 public class Errors {  
2     private Errors e = null;  
3     private Class<? extends String> c = String.getClass();  
4  
5     public Errors(Errors ee) { e = ee; }  
6     public Errors() { this(this); }  
7  
8     public boolean f() {  
9         Class<?> old_c = c;  
10        c = Object.class;  
11        return (old_c == c);  
12    }  
13 }
```

244. (2007-6-29)

Individuare gli errori di compilazione nella seguente classe. Commentare brevemente ciascun errore e fornire una possibile correzione.

```
1 public class Errors {  
2     private static int num = 7;  
3     private Integer z = 8;  
4     Map<Integer, Errors> m = new Map<Integer, Errors>();  
5  
6     public Errors() {}  
7  
8     private static class A {  
9         private A() { num += z; }  
10    }  
11    private void f() {  
12        m.put(7, new Errors() { public int g() { return 0; } });  
13    }  
14  
15    public static final A a = new A();  
16 }
```

245. (2007-4-26)

Individuare gli errori di compilazione nel seguente programma. Commentare brevemente ciascun errore e fornire una possibile correzione.

```
1 public class Errors {  
2     private static int sval = 7;  
3     private int val = sval;  
4  
5     public Errors() { super(); }  
6  
7     private class A {  
8         private A(int n) { val += n; }  
9     }  
10    private class B extends A {
```

6 Trova l'errore

```
11     B() { val = sval; }
12 }
13
14 public static void main(String[] args) {
15     Errors t = new Errors;
16     A a = t.new A(5);
17     B b = a.new B();
18 }
19 }
```

246. (2006-9-15)

Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma. Il programma dovrebbe lanciare un nuovo thread che stampa gli interi da 0 a 9.

```
1 class Test extends Runnable {
2     private Thread thread;
3
4     public Test() {
5         thread = new Thread();
6     }
7
8     public run() {
9         int i = 0;
10        for (i=0; i<10 ;i++)
11            System.out.println("i=" + i);
12    }
13
14    public static void main(String args[]) {
15        Test t = new Test();
16        t.start();
17    }
18 }
```

247. (2006-7-17)

Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma.

```
1 class Test {
2     Collection<?> c;
3
4     public int f(Collection<? extends Number> c) {
5         return c.size();
6     }
7
8     public void g(Set<? extends Number> c) {
9         this.c = c;
10    }
11
12    private <T extends Number> T myAdd(T x) {
13        c.add(x);
14        return x;
15    }
16
17    public static void main(String args[]) {
18        Test t = new Test();
19
20        t.f(new LinkedList<Integer>());
21        t.g(new ArrayList<Integer>());
22        t.myAdd(new Double(2.0));
23    }
24 }
```

248. (2006-6-26)

Individuare e descrivere sinteticamente gli errori nel seguente programma.

```
1 class Test {  
2     public static void f(List<? extends Number> l) {  
3         l.add(new Integer(3));  
4     }  
5     public static <T> T myGet(Map<T, ? extends T> m, T x) {  
6         return m.get(x);  
7     }  
8  
9     public static void main(String args[]) {  
10        f(new LinkedList<Integer>());  
11        f(new ArrayList<Boolean>());  
12        f(new List<Double>());  
13        Object o = myGet(new HashMap<Number, Integer>(), new Integer(7));  
14    }  
15 }
```

249. (2006-4-27)

Individuare e correggere gli errori nel seguente programma.

```
/*  
 * Questo programma somma due numeri costanti forniti  
 * staticamente da programma e ne stampa il risultato.  
 */  
  
public class SommaDueNumeri{  
    public void main(String[] args){  
        System.out.print("Questo programma somma due numeri.),  
        i = 185;  
        int j = 1936.27; // tasso di conversione lire in evri :-)  
        System.out.print("La somma di " + i + " e " + j + " e': ");  
        System.out.println(i+j);  
    }  
}
```


7 Design by contract

250. (Merge, 2018-5-2)

Realizzare un metodo chiamato `merge` che rispetti il seguente contratto:

Pre-condizione Accetta due `LinkedList` dello stesso tipo e di pari lunghezza.

Post-condizione Restituisce una nuova `LinkedList` ottenuta alternando gli elementi della prima lista e quelli della seconda.

Ad esempio, se la prima lista contiene (1, 2, 3) e la seconda lista (4, 5, 6), la nuova lista deve contenere (1, 4, 2, 5, 3, 6).

Penale Se le liste non hanno la stessa lunghezza, lancia `IllegalArgumentException`.

251. (Count, 2016-4-21)

Il metodo `count` accetta una `LinkedList` e restituisce un intero. Il suo contratto è il seguente:

pre-condizione La lista contiene stringhe.

post-condizione Restituisce la somma delle lunghezze delle stringhe presenti nella lista.

Dire quali dei seguenti sono contratti validi per un overriding di `f`, motivando la risposta.

a) **pre-condizione** Nessuna.

post-condizione Restituisce la somma delle lunghezze delle stringhe presenti nella lista (oggetti diversi da stringhe vengono ignorati).

b) **pre-condizione** La lista contiene stringhe non vuote.

post-condizione Restituisce la lunghezza della lista.

8 Programmazione parametrica (generics)

252. (Accumulator, 2020-2-27)

Realizzare la classe parametrica `Accumulator`, che accetta come parametro di tipo una sottoclasse di `Number` e offre i seguenti servizi:

- inserimento di un numero (metodo `add`);
- scorrimento di tutti i numeri inseriti fino a quel momento, divisi tra negativi e non (metodi `negatives` e `positives`);
- somma di tutti i numeri inseriti fino a quel momento (metodo `sum`).

Suggerimento: Si ricordi che la classe `Number` prevede il metodo `double doubleValue()`.

L'implementazione deve rispettare il seguente esempio d'uso.

| Esempio d'uso: | Output: |
|---|------------------------------|
| <code>Accumulator<Integer> acc1 = new Accumulator<>(); acc1.add(10); acc1.add(42); acc1.add(-5); acc1.add(10); for (Integer n: acc1.positives()) System.out.println(n); for (Integer n: acc1.negatives()) System.out.println(n);</code> | 10 42 10 -5 32.0 |
| <code>Accumulator<Double> acc2 = new Accumulator<>(); acc2.add(-10.0); acc2.add(42.0); System.out.println(acc2.sum());</code> | |

Invece, ciascuna delle seguenti due istruzioni deve provocare un errore di compilazione:

```
acc1.positives().add(5);  
Accumulator<String> acc3 = new Accumulator<>();
```

253. (Range, 2019-2-15)

Realizzare la classe parametrica `Range`, che rappresenta un intervallo di oggetti dotati di ordinamento naturale, con le seguenti funzionalità:

- Il costruttore accetta gli estremi dell'intervallo (l'oggetto minimo e l'oggetto massimo).
- Il metodo `isInside` accetta un oggetto `x` e restituisce `true` se e solo se `x` appartiene all'intervallo.
- Il metodo `isOverlapping` accetta un altro intervallo `x` e restituisce `true` se e solo se `x` è sovrapposto a questo intervallo (cioè se i due hanno intersezione non vuota).
- Il metodo `equals` è ridefinito in modo che due intervalli con gli stessi estremi risultino uguali.
- Il metodo `hashCode` è ridefinito in modo da essere coerente con `equals`.

Porre attenzione alla firma di `isOverlapping` e spiegare se è completa o meno.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Range<Integer> a = new Range<>(10, 20);
System.out.println(a.isInside(10));
System.out.println(a.isInside(50));

Range<String> b = new Range<>("albero", "dirupo"),
c = new Range<>("casa", "catrame");
System.out.println(b.isOverlapping(c));

Range<Object> d = new Range<>(); // errore di compilazione
```

Output:
true
false
true

254. (Generic constructor, 2017-4-26)

Ipotizzando la disponibilità delle classi Person, Employee e Manager, ciascuna sottoclasse della precedente, realizzare la classe Container in modo che il seguente frammento sia corretto:

```
Container<Employee> cont1 = new <String>Container<Employee>("ciao");
Container<Employee> cont2 = new <Integer>Container<Employee>(new Integer(42));
Container<Manager> cont3 = new <Integer>Container<Manager>(new Integer(42));
```

e ciascuna delle seguenti istruzioni provochi un errore di compilazione:

```
Container<Employee> cont4 = new <Object>Container<Employee>(new Object());
Container<Person> cont5 = new <Integer>Container<Person>(new Integer(42));
```

255. (2016-7-21)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A extends B<Object> {
    private B<?> b;
    private String msg;
    public A() {
        b = new B<Object>(null);
        msg = B.<A>buildMessage(this);
    }
    public Set<? super Number> f(Set<Integer> set1, Set<String> set2) {
        for (Integer n: b)
            if (b.check(set1, n))
                return b.process(set1, set2, n);
        return b.process(set2, set1, null);
    }
}
```

256. (2016-1-27)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A
{
    public static <S,T extends S> void f(Set<S> set1, Set<T> set2)
    {
        B.process(set1, set2);
        B.process(set2, set1);

        B<S> b = new B<S>();

        S choice1 = b.select(set1),
        choice2 = b.select(set2);

        Collection<? extends S> c = b.filter(set1);
```

```

        HashSet<? super S> hs = b.filter(set1);
    }
}

```

257. (Relation, 2015-1-20)

Realizzare la classe Relation, che rappresenta una relazione binaria tra un insieme S e un insieme T. In pratica, una Relation è analoga ad una Map, con la differenza che la Relation accetta chiavi duplicate.

Il metodo put aggiunge una coppia di oggetti alla relazione. Il metodo remove rimuove una coppia di oggetti dalla relazione. Il metodo image accetta un oggetto x di tipo S e restituisce l'insieme degli oggetti di tipo T che sono in relazione con x. Il metodo preImage accetta un oggetto x di tipo T e restituisce l'insieme degli oggetti di tipo S che sono in relazione con x.

Esempio d'uso:

```

Relation<Integer,String> r = new Relation<Integer,String>();
r.put(0, "a"); r.put(0, "b"); r.put(0, "c");
r.put(1, "b"); r.put(2, "c");
r.remove(0, "a");
Set<String> set0 = r.image(0);
Set<Integer> setb = r.preImage("b");
System.out.println(set0);
System.out.println(setb);

```

Output:

[b, c]
[0, 1]

258. (Pair, 2013-4-29)

Realizzare la classe parametrica Pair, che rappresenta una coppia di oggetti di tipo potenzialmente diverso. La classe deve supportare le seguenti funzionalità:

- 1) due Pair sono uguali secondo equals se entrambe le loro componenti sono uguali secondo equals;
- 2) il codice hash di un oggetto Pair è uguale allo XOR tra i codici hash delle sue due componenti;
- 3) la stringa corrispondente ad un oggetto Pair è “(str1,str2)”, dove str1 (rispettivamente, str2) è la stringa corrispondente alla prima (risp., seconda) componente.

Esempio d'uso:

```

Pair<String,Integer> p1 = new Pair<String,Integer>("uno", 1);
System.out.println(p1);

```

Output:

(uno, 1)

259. (BoundedMap, 2012-6-18)

Implementare la classe BoundedMap, che rappresenta una mappa con capacità limitata. Il costruttore accetta la dimensione massima della mappa. I metodi get e put sono analoghi a quelli dell'interfaccia Map. Se però la mappa è piena e viene invocato il metodo put con una chiave nuova, verrà rimossa dalla mappa la chiave che fino a quel momento è stata ricercata meno volte con get.

L'implementazione deve rispettare il seguente caso d'uso.

Esempio d'uso:

```

BoundedMap<String,String> m = new BoundedMap<String,String>(2);
m.put("NA", "Napoli");
m.put("SA", "Salerno");
System.out.println(m.get("NA"));
m.put("AV", "Avellino");
System.out.println(m.get("SA"));

```

Output dell'esempio d'u-

so:
Napoli
null

260. (2011-3-4)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A<T extends Cloneable> extends B<T> {
    private Cloneable t, u;
    private B<String> b;
    private int i;

    public A(T x) {
        t = x;
        u = g1();
        b = g2(x);
        i = this.compareTo("ciao");
    }

    public Double f(Object o) {
        Number n = super.f(o);
        if (n instanceof Double) return (Double)n;
        else return null;
    }
}
```

261. (MakeMap, 2011-2-7)

Scrivere un metodo che accetta due liste (List) k e v di pari lunghezza, e restituisce una mappa in cui all'elemento i -esimo di k viene associato come valore l'elemento i -esimo di v .

Il metodo lancia un'eccezione se le liste non sono di pari lunghezza, oppure se k contiene elementi duplicati.

Si ricordi che non è opportuno utilizzare l'accesso posizionale su liste generiche.

262. (Intersect, 2010-9-14)

Implementare il metodo statico `intersect`, che accetta come argomenti due Collection x e y e restituisce una nuova Collection che contiene l'intersezione di x ed y (cioè, gli oggetti comuni ad entrambe le collezioni).

Prestare particolare attenzione alla scelta della firma del metodo.

263. (2010-7-26)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private B<Integer> b1 = new B<Integer>(null);
    private B<?> b2 = B.f(3);
    private Comparable<? extends Number> c = new B<Double>();

    public Object f() {
        Integer x = b1.getIt();
        Integer y = x + b2.getIt();
        return new B<String>(new A());
    }
}
```

264. (SelectKeys, 2010-11-30)

Scrivere un metodo che accetta una lista l e una mappa m , e restituisce una nuova lista che contiene gli elementi di l che compaiono come chiavi in m . Porre particolare attenzione alla scelta della firma.

265. (2009-7-9)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {  
    public interface Convertible<T> {  
        public T convert();  
    }  
    private Convertible<A> x = new B();  
    private Iterable<A> y = new B(3);  
  
    private Iterable<A> z = B.g(x);  
    private Iterable<? extends B> t = B.g(B.b);  
}
```

266. (2009-6-19)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {  
    private List<? extends String> l = B.getList();  
  
    public <T> void f(T x, Comparator<? super T> y) {  
        y.compare(x, B.getIt(x));  
    }  
    public void g(Set<? super Integer> s) {  
        Set<String> s2 = B.convert();  
        f(new B(), B.something());  
        f(new Integer(4), B.something());  
    }  
}
```

267. (Interleave, 2009-2-19)

Implementare un metodo statico `interleave` che prende come argomento tre `LinkedList`: *A*, *B* e *C*. Senza modificare *A* e *B*, il metodo aggiunge tutti gli elementi di *A* e di *B* a *C*, in modo alternato (prima un elemento di *A*, poi uno di *B*, poi un altro elemento di *A*, e così via). Porre particolare attenzione alla scelta della firma di `interleave`, in modo che sia la più generale possibile, ma senza utilizzare parametri di tipo inutili.

268. (Split, 2009-1-29)

Implementare un metodo statico `split` che prende come argomento tre collezioni *A*, *B* e *C*. Senza modificare *A*, il metodo inserisce metà dei suoi elementi in *B* e l'altra metà in *C*. Porre particolare attenzione alla scelta della firma di `split`, in modo che sia la più generale possibile, ma senza utilizzare parametri di tipo inutili.

269. (BoolExpr, 2008-2-25)

La classe (o interfaccia) `BoolExpr` rappresenta un'espressione dell'algebra booleana (ovvero un circuito combinatorio). Il tipo più semplice di espressione è una semplice variabile, rappresentata dalla classe `BoolVar`, sottotipo di `BoolExpr`. Espressioni più complesse si ottengono usando gli operatori di tipo *and*, *or* e *not*, corrispondenti ad altrettante classi sottotipo di `BoolExpr`. Tutte le espressioni hanno un metodo `eval` che, dato il valore assegnato alle variabili, restituisce il valore dell'espressione. Si consideri *attentamente* il seguente caso d'uso.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>public static void main(String args[]) { BoolVar x = new BoolVar("x"); BoolVar y = new BoolVar("y"); BoolExpr notx = new BoolNot(x); BoolExpr ximpliesy = new BoolOr(notx, y); Map<BoolVar,Boolean> m = new HashMap<BoolVar,Boolean>() ; m.put(x, true); m.put(y, true); System.out.println(x.eval(m)); System.out.println(ximpliesy.eval(m)); m.put(y, false); System.out.println(ximpliesy.eval(m)); }</pre> | <pre>true true false</pre> |

270. (MyFor, 2008-2-25)

Implementare una classe **MyFor** in modo che, per tutti i numeri interi *a*, *b* e *c*, il ciclo:

```
for (Integer i: new MyFor(a, b, c)) { ... }
```

sia equivalente al ciclo:

```
for (Integer i=a; i<b ; i+=c) { ... }
```

271. (Sorter, 2008-1-30)

Implementare una classe parametrica **Sorter**, con un solo metodo **check**. Il metodo **check** confronta l'oggetto che riceve come argomento con quello che ha ricevuto alla chiamata precedente, o con quello passato al costruttore se si tratta della prima chiamata a **check**. Il metodo restituisce -1 se il nuovo oggetto è più piccolo del precedente, 1 se il nuovo oggetto è più grande del precedente e 0 se i due oggetti sono uguali. Per effettuare i confronti, **Sorter** si basa sul fatto che il tipo usato come parametro implementi l'interfaccia **Comparable**.

| | |
|--|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Sorter<Integer> s = new Sorter<Integer>(7); System.out.println(s.check(4)); System.out.println(s.check(1)); System.out.println(s.check(6)); System.out.println(s.check(6));</pre> | <pre>-1 -1 1 0</pre> |

272. (Selector, 2007-9-17)

L'interfaccia parametrica **Selector** prevede un metodo **select** che restituisce un valore booleano per ogni elemento del tipo parametrico.

```
public interface Selector<T> {
    boolean select(T x);
}
```

Implementare una classe **SelectorIterator** che accetta una collezione e un selettore dello stesso tipo, e permette di iterare sugli elementi della collezione per i quali il selettore restituisce true.

| | |
|---|---|
| Esempio d'uso: <pre>Integer [] a = { 1, 2, 45, 56, 343, 22, 12, 7, 56}; List<Integer> l = Arrays.asList(a); Selector<Integer> pari = new Selector<Integer>() { public boolean select(Integer n) { return (n % 2) == 0; } }; for (Integer n: new SelectorIterator<Integer>(l, pari)) System.out.print(n + " ");</pre> | Output dell'esempio d'uso: 2 56 22 12 56 |
|---|---|

273. (FunnyOrder, 2007-9-17)

Determinare l'output del seguente programma e descrivere brevemente l'ordinamento dei numeri interi definito dalla classe FunnyOrder.

```
public class FunnyOrder implements Comparable<FunnyOrder> {
    private int val;
    public FunnyOrder(int n) { val = n; }
    public int compareTo(FunnyOrder x) {
        if (val%2 == 0 && x.val%2 != 0) return -1;
        if (val%2 != 0 && x.val%2 == 0) return 1;
        if (val < x.val) return -1;
        if (val > x.val) return 1;
        return 0;
    }
    public static void main(String[] args) {
        List<FunnyOrder> l = new LinkedList<FunnyOrder>();
        l.add(new FunnyOrder(16));
        l.add(new FunnyOrder(3));
        l.add(new FunnyOrder(4));
        l.add(new FunnyOrder(10));
        l.add(new FunnyOrder(2));
        Collections.sort(l);
        for (FunnyOrder f: l)
            System.out.println(f.val + " ");
    }
}
```

274. (CommonDividers, 2007-7-20)

Implementare una classe CommonDividers che rappresenta tutti i divisori comuni di due numeri interi, forniti al costruttore. Su tale classe si deve poter iterare secondo il seguente caso d'uso. Dei 30 punti, 15 sono riservati a coloro che realizzzeranno l'iteratore senza usare spazio aggiuntivo. Viene valutato positivamente l'uso di classi anonime laddove opportuno.

| | |
|---|--|
| Esempio d'uso: <pre>for (Integer n: new CommonDividers(12, 60)) System.out.print(n + " ");</pre> | Output dell'esempio d'uso: 1 2 3 4 6 12 |
|---|--|

275. (ParkingLot, 2007-7-20)

Implementare una classe ParkingLot, che rappresenta un parcheggio con posti auto disposti secondo una griglia $m \times n$. Il costruttore prende come argomenti le dimensioni m ed n del parcheggio. Il metodo carIn aggiunge un veicolo al parcheggio e restituisce la riga e la colonna del posto assegnato al nuovo veicolo, oppure null se il parcheggio è pieno. Il metodo carOut prende come argomenti le coordinate di un veicolo che sta lasciando il parcheggio e restituisce il numero di secondi trascorsi dal veicolo nel parcheggio, oppure null se alle coordinate indicate non si trova alcun veicolo.

Suggerimento: utilizzare la classe java.util.Date per misurare il tempo.

Esempio d'uso:

```
ParkingLot p = new ParkingLot(10, 10);

Pair<Integer> pos1 = p.carIn();
Pair<Integer> pos2 = p.carIn();
Thread.sleep(1000);
int sec1 = p.carOut(pos1);
Thread.sleep(1000);
int sec2 = p.carOut(pos2);

System.out.println("(" + pos1.getFirst() + ", " + pos1.getSecond() + "), " + sec1
    );
System.out.println("(" + pos2.getFirst() + ", " + pos2.getSecond() + "), " + sec2
    );
```

Output:
 $(0, 0), 1$
 $(0, 1), 2$

276. (2007-7-20)

Individuare gli errori di *compilazione* nella seguente classe. Commentare brevemente ciascun errore e fornire una possibile correzione.

```
1 public class Errors {
2     private Errors e = null;
3     private Class<? extends String> c = String.getClass();
4
5     public Errors(Errors ee) { e = ee; }
6     public Errors()           { this(this); }
7
8     public boolean f() {
9         Class<?> old_c = c;
10        c = Object.class;
11        return (old_c == c);
12    }
13 }
```

277. (Polinomio su un campo generico, 2007-6-29)

Un *campo* (field) è una struttura algebrica composta da un insieme detto supporto, dalle due operazioni binarie di somma e prodotto, e dai due elementi neutri, per la somma e per il prodotto rispettivamente. La seguente interfaccia rappresenta un campo con supporto T:

```
public interface Field<T> {
    T plus(T x, T y); // la somma
    T times(T x, T y); // il prodotto
    T getOne();        // restituisce l'elemento neutro per il prodotto
    T getZero();       // restituisce l'elemento neutro per la somma
}
```

- (10 punti) Implementare una classe DoubleField che implementi Field<Double>.
- (30 punti) Implementare una classe Polynomial che rappresenti un polinomio con coefficienti in un dato campo. Il costruttore accetta un array di coefficienti e il campo sul quale interpretare i coefficienti. Il metodo eval restituisce il valore del polinomio per un dato valore della variabile.

Esempio d'uso:

```
Double[] d = { 2.0, 3.0, 1.0 }; // 2 + 3 x + x^2
Polynomial<Double> p = new Polynomial<Double>(d, new
    DoubleField());

System.out.println(p.eval(3.0));
System.out.println(p.eval(2.0));
```

Output:
 20.0
 12.0

278. (**Polinomio bis, 2007-2-7**)

Si consideri la seguente classe `Pair`.

```
public class Pair<T, U>
{
    public Pair(T first, U second) { this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public U getSecond() { return second; }

    private T first;
    private U second;
}
```

Un *polinomio* è una espressione algebrica del tipo $a_0 + a_1x + \dots + a_nx^n$. Si implementi una classe `Polynomial`, dotata di un costruttore che accetta un array contenente i coefficienti $a_0 \dots a_n$. Deve essere possibile iterare sulle coppie (esponente, coefficiente) in cui il coefficiente è diverso da zero. Cioè, `Polynomial` deve implementare `Iterable<Pair<Integer, Double>>`. Infine, il metodo `toString` deve produrre una stringa simile a quella mostrata nel seguente caso d'uso.

| Esempio d'uso: | Output dell'esempio d'uso: |
|---|--|
| <pre>double a1[] = {1, 2, 0, 3}; double a2[] = {0, 2}; Polynomial p1 = new Polynomial(a1); Polynomial p2 = new Polynomial(a2); System.out.println(p1); System.out.println(p2); for (Pair<Integer, Double> p: p1) System.out.println(p.getFirst() + " : " + p.getSecond());</pre> | <pre>1.0 + 2.0 x^1 + 3.0 x^3 2.0 x^1 0 : 1.0 1 : 2.0 3 : 3.0</pre> |

279. (**Inventory, 2007-2-23**)

Definire una classe parametrica `Inventory<T>` che rappresenta un inventario di oggetti di tipo `T`. Il costruttore senza argomenti crea un inventario vuoto. Il metodo `add` aggiunge un oggetto di tipo `T` all'inventario. Il metodo `count` prende come argomento un oggetto di tipo `T` e restituisce il numero di oggetti uguali all'argomento presenti nell'inventario. Infine, il metodo `getMostCommon` restituisce l'oggetto di cui è presente il maggior numero di esemplari.

| Esempio d'uso: | Output dell'esempio d'uso: |
|---|------------------------------|
| <pre>Inventory<Integer> a = new Inventory<Integer>(); Inventory<String> b = new Inventory<String>(); a.add(7); a.add(6); a.add(7); a.add(3); b.add("ciao"); b.add("allora?"); b.add("ciao_ciao"); b.add("allora?"); "); System.out.println(a.count(2)); System.out.println(a.count(3)); System.out.println(a.getMostCommon()); System.out.println(b.getMostCommon());</pre> | <pre>so: 0 1 7 allora?</pre> |

280. (**2006-7-17**)

Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma.

```
1 class Test {
2     Collection<?> c;
3
4     public int f(Collection<? extends Number> c) {
5         return c.size();
```

```

6     }
7
8     public void g(Set<? extends Number> c) {
9         this.c = c;
10    }
11
12    private <T extends Number> T myAdd(T x) {
13        c.add(x);
14        return x;
15    }
16
17    public static void main(String args[]) {
18        Test t = new Test();
19
20        t.f(new LinkedList<Integer>());
21        t.g(new ArrayList<Integer>());
22        t.myAdd(new Double(2.0));
23    }
24 }
```

281. (2006-6-26)

Individuare e descrivere sinteticamente gli errori nel seguente programma.

```

1 class Test {
2     public static void f(List<? extends Number> l) {
3         l.add(new Integer(3));
4     }
5     public static <T> T myGet(Map<T, ? extends T> m, T x) {
6         return m.get(x);
7     }
8
9     public static void main(String args[]) {
10        f(new LinkedList<Integer>());
11        f(new ArrayList<Boolean>());
12        f(new List<Double>());
13        Object o = myGet(new HashMap<Number, Integer>(), new Integer(7));
14    }
15 }
```

9 Classe mancante

282. (2019-10-9)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {  
    Comparator<Double> b = new B(null);  
  
    Comparator<String> c = (x, y) -> B.g(x, y);  
  
    <T> A f(T x, T y) {  
        return new B(x==y);  
    }  
}
```

283. (2016-7-21)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A extends B<Object> {  
    private B<?> b;  
    private String msg;  
    public A() {  
        b = new B<Object>(null);  
        msg = B.<A>buildMessage(this);  
    }  
    public Set<? super Number> f(Set<Integer> set1, Set<String> set2) {  
        for (Integer n: b)  
            if (b.check(set1, n))  
                return b.process(set1, set2, n);  
        return b.process(set2, set1, null);  
    }  
}
```

284. (2016-1-27)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A  
{  
    public static <S,T extends S> void f(Set<S> set1, Set<T> set2)  
    {  
        B.process(set1, set2);  
        B.process(set2, set1);  
  
        B<S> b = new B<S>();  
  
        S choice1 = b.select(set1),  
            choice2 = b.select(set2);  
  
        Collection<? extends S> c = b.filter(set1);  
        HashSet<? super S> hs = b.filter(set1);  
    }  
}
```

9 Classe mancante

285. (2011-3-4)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A<T extends Cloneable> extends B<T> {
    private Cloneable t, u;
    private B<String> b;
    private int i;

    public A(T x) {
        t = x;
        u = g1();
        b = g2(x);
        i = this.compareTo("ciao");
    }

    public Double f(Object o) {
        Number n = super.f(o);
        if (n instanceof Double) return (Double)n;
        else return null;
    }
}
```

286. (2010-7-26)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private B<Integer> b1 = new B<Integer>(null);
    private B<?> b2 = B.f(3);
    private Comparable<? extends Number> c = new B<Double>();

    public Object f() {
        Integer x = b1.getIt();
        Integer y = x + b2.getIt();
        return new B<String>(new A());
    }
}
```

287. (2010-1-22)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A extends B {
```

```
    public A() {
        b1 = new B.C(true);
        b2 = new B(false);
    }
```

```
    public B f(Object o) {
        B x = super.f(o);
        return x.clone();
    }
```

```
    private B.C c = new B.C(3);
    private B b1, b2;
}
```

288. (2009-9-l'8)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    private A a = new B.C(3);
    private double x = a.f(3);
    private B b = new B.D(3);

    private int f(int n) {
        g(new B(3), n);
        return 2*n;
    }
    private void g(A u, int z) { }
    private void g(B u, double z) { }

    public A(int i) { }
}

```

289. (2009-7-9)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    public interface Convertible<T> {
        public T convert();
    }
    private Convertible<A> x = new B();
    private Iterable<A> y = new B(3);

    private Iterable<A> z = B.g(x);
    private Iterable<? extends B> t = B.g(B.b);
}

```

290. (2009-6-19)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    private List<? extends String> l = B.getList();

    public <T> void f(T x, Comparator<? super T> y) {
        y.compare(x, B.getIt(x));
    }
    public void g(Set<? super Integer> s) {
        Set<String> s2 = B.convert(s);
        f(new B(), B.something);
        f(new Integer(4), B.something);
    }
}

```

291. (2009-4-23)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    private B b1 = new B(null);
    private B b2 = new B.C();
    private B b3 = b1.new D();

    private int f(Object x) {
        if (x==null) throw b2;

        long l = b1.g();
        return b1.g();
}

```

```

    }
}
```

292. (2009-2-19)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    private Comparator<Double> b = new B(null);

    private <T> A f(T x, T y) {
        return new B(x==y);
    }
}
```

293. (2009-11-27)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    public static final A a = new B(null);
    public final int n = B.f(3);

    public Object g() {
        B b = new B();
        B.C c = b.new C(7);
        return c;
    }

    public A(int i) { }
}
```

294. (2009-1-15)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente, permetta la compilazione della classe A e produca l'output indicato.

Inoltre, rispondere alle seguenti domande:

- Quale design pattern si ritrova nel metodo `Collections.sort`?
- Quale ordinamento sui numeri interi realizza la vostra classe B?

| | Output richiesto: |
|---|-------------------|
| <code>public class A {</code> | 20 |
| <code> public static void main(String[] args) {</code> | 50 |
| <code> List<Integer> l = new LinkedList<Integer>();</code> | 70 |
| <code> l.add(3); l.add(70); l.add(23); l.add(50); l.add(5); l.add</code> | 3 |
| <code> (20);</code> | 5 |
| <code> Collections.sort(l, new B());</code> | 23 |
| <code> for (Integer i: l)</code> | |
| <code> System.out.println(i);</code> | |
| <code>}</code> | |

295. (2008-7-9)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A<T> {
    private B myb = new B(null);

    private int f(T x) {
        Iterator<?> i = myb.new MyIterator();
        String msg = B.f(x);
        double d = myb.g();
        return myb.g();
    }
}

```

296. (2008-6-19)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A<T> {
    private B<?> myb = new B<Integer>();

    private Integer f(T x) {
        T y = myb.copia(x);
        List<? extends Number> l = B.lista();
        int i = myb.f(2);
        boolean b = myb.f(2.0);
        return myb.g();
    }
}

```

297. (2008-3-27)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A extends B {
    public A(int x) {
        super(x-1, x / 2.0);
    }
    public A(double inutile) { }

    private void stampa(String s) {
        if (s == null) throw new B(s);
        else System.out.println(s);
    }
}

```

298. (2008-2-25)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    private B myb;

    private B f(B b) {
        myb = new B(true + "true");
        int x = b.confronta(myb);
        int y = myb.confronta(b);
        return myb.valore();
    }

    private Object zzz = B.z;
}

```

299. (2008-1-30)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {  
    private B myb;  
  
    private int f(B b) {  
        A x = B.copia(b);  
        myb = B.copia(77);  
        double d = myb.g();  
        return myb.g();  
    }  
  
    private int x = B.x;  
}
```

10 Classi interne

300. (InternalLayout4, 2022-3-28)

Date le seguenti classi:

```
public class A {  
    private int n = 0;  
    public A() {  
        n++;  
    }  
    public static class B {  
        private int i = 1;  
    }  
    public Object makeObj(int val) {  
        return new B() {  
            private int j = val;  
        };  
    }  
}
```

Disegnare il *memory layout* che risulta al termine dell'esecuzione del seguente frammento di codice, evidenziando gli eventuali riferimenti impliciti, le variabili catturate e i loro valori:

```
A a1 = new A();  
A a2 = new A();  
Object x = a1.makeObj(42);  
A.B y = (A.B) a1.makeObj(42);  
A.B b = new A.B();
```

301. (InternalLayout3, 2022-2-24)

Date le seguenti classi:

```
public class A {  
    private int n = 0;  
    A() {  
        n++;  
    }  
    public static class B {  
        int i = 1;  
    }  
    public class C {  
        int j = 2;  
    }  
}
```

Disegnare il *memory layout* che risulta al termine dell'esecuzione del seguente frammento di codice (si supponga che il frammento si trovi nello stesso pacchetto della classe A):

```
A a1 = new A();  
A a2 = new A();  
A.B b = new A.B();  
A.C c1 = a1.new C();  
A.C c2 = a2.new C();
```

302. (InternalLayout2, 2021-9-24)

Date le seguenti classi:

```
class A {
    public int n = 0;
    private C c = new C();
    public static class B {
        int i = 1;
    }
    public class C {
        int j = 2;
        public C() {
            A.this.n++;
        }
    }
}
```

Disegnare il *memory layout* alla fine dell'esecuzione del seguente frammento di codice:

```
A a = new A();
A.B b = new A.B();
A.C c = a.new C();
```

303. (InternalLayout1, 2021-7-26)

Date le seguenti classi:

```
public class A {
    public int n;
    private A myself = this;
    public static class B extends A {
        int i = 1;
    }
    public class C {
        int j = 2;
    }
}
```

Disegnare il *memory layout* del seguente frammento di codice:

```
A a1 = new A();
A a2 = new A.B();
A.C c1 = a1.new C();
A.C c2 = a2.new C();
```

304. (Microwave, 2019-9-20)

L'interfaccia `Microwave` rappresenta i controlli di un forno a microonde, con un metodo per accenderlo impostando la potenza (in watt) e un metodo per spegnerlo.

```
interface Microwave {
    void on(int power);
    void off();
}
```

Implementare la classe `Program`, che rappresenta una sequenza di istruzioni per un forno a microonde, rispettando il seguente caso d'uso.

La seguente riga crea un programma che cuoce per 10 secondi a 500 watt e poi cuoce per altri 5 secondi a 700 watt.

```
Program p = Program.make().on(500).delay(10000).on(700).delay(5000).off();
```

Dato un oggetto `oven` di tipo `Microwave`, la seguente riga esegue il programma `p` sul forno in questione, invocando i metodi `on` e `off` dell'oggetto `oven` al momento giusto.

```
p.executeOn(oven);
```

Il metodo `executeOn` è bloccante e restituisce il controllo al chiamante solo quando il programma è terminato.

305. (Studente, 2018-3-23)

Implementare la classe `Studente` e le due sottoclassi `Triennale` e `Magistrale`. Uno studente è caratterizzato da nome e matricola. Ciascuna sottoclasse ha un prefisso che viene aggiunto a tutte le sue matricole. Due studenti sono considerati uguali da `equals` se hanno lo stesso nome e la stessa matricola (compreso il prefisso).

L'implementazione deve rispettare il seguente esempio d'uso:

| | |
|--|---------------------------------------|
| Esempio d'uso: | Output: |
| <pre>Studente.Triennale.setPrefisso ("N86"); Studente.Magistrale.setPrefisso ("N97"); Object luca1 = new Studente.Triennale("Luca", "004312"), luca2 = new Studente.Triennale("Luca", "004312"), anna1 = new Studente.Triennale("Anna", "004312"), anna2 = new Studente.Magistrale("Anna", "004312"); System.out.println(luca1.equals(luca2)); System.out.println(anna1.equals(anna2)); System.out.println(anna1);</pre> | <pre>true false Anna N86/004312</pre> |

306. (Engine, 2016-4-21)

[CROWDGRADER] Realizzare la classe `Engine`, che rappresenta un motore a combustione, caratterizzato da cilindrata (in cm³) e potenza (in cavalli). Normalmente, due oggetti `Engine` sono uguali se hanno la stessa cilindrata e la stessa potenza. Il metodo `byVolume` converte questo `Engine` in modo che venga confrontata solo la cilindrata. Analogamente, il metodo `byPower` converte questo `Engine` in modo che venga confrontata solo la potenza.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|--|
| Esempio d'uso: | Output: |
| <pre>Engine a = new Engine(1200, 69), b = new Engine(1200, 75), c = new Engine(1400, 75); System.out.println(a); System.out.println(a.equals(b)); Engine aVol = a.byVolume(), bVol = b.byVolume(); System.out.println(aVol); System.out.println(aVol.equals(bVol)); System.out.println(a==aVol); Engine bPow = b.byPower(), cPow = c.byPower(); System.out.println(bPow); System.out.println(bPow.equals(cPow));</pre> | <pre>(1200.0 cm3, 69.0 CV) false (1200.0 cm3, 69.0 CV) true false (1200.0 cm3, 75.0 CV) true</pre> |

307. (Curriculum, 2016-1-27)

Un oggetto `Curriculum` rappresenta una sequenza di lavori, ognuno dei quali è un'istanza della classe `Job`. Il costruttore di `Curriculum` accetta il nome di una persona. Il metodo `addJob` aggiunge un lavoro alla sequenza, caratterizzato da una descrizione e dall'anno di inizio, restituendo un nuovo oggetto di tipo `Job`. Infine, la classe `Job` offre il metodo `next`, che restituisce *in tempo costante* il lavoro successivo nella sequenza (oppure `null`).

Implementare le classi `Curriculum` e `Job`, rispettando il seguente caso d'uso.

Caso d'uso:

```
Curriculum cv = new Curriculum("Walter_White");
Curriculum.Job j1 = cv.addJob("Chimico", 1995);
Curriculum.Job j2 = cv.addJob("Insegnante", 2005);
Curriculum.Job j3 = cv.addJob("Cuoco", 2009);
```

```
System.out.println(j2.next());
System.out.println(j3.next());
```

Output:

```
Cuoco: 2009
null
```

308. (Controller, 2015-6-24)

Realizzare la classe **Controller**, che rappresenta una centralina per autoveicoli, e la classe **Function**, che rappresenta una funzionalità del veicolo, che può essere accesa o spenta. Alcune funzionalità sono *incompatibili* tra loro, per cui accenderne una fa spegnere automaticamente l'altra.

La classe Controller ha due metodi: **addFunction** aggiunge al sistema una nuova funzionalità con un dato nome; **printOn** stampa a video i nomi delle funzionalità attive. La classe Function ha tre metodi: **turnOn** e **turnOff** per attivarla e disattivarla; **setIncompatible** accetta un'altra funzionalità *x* e imposta un'incompatibilità tra *this* e *x*.

Leggere attentamente il seguente caso d'uso, che mostra, tra le altre cose, che l'incompatibilità è automaticamente simmetrica, ma *non* transitiva.

Caso d'uso:

```
Controller c = new Controller();
Controller.Function ac = c.addFunction("Aria_condizionata");
Controller.Function risc = c.addFunction("Riscaldamento");
Controller.Function sedile = c.addFunction("Sedile_riscaldato");

ac.setIncompatible(risc);
ac.setIncompatible(sedile);

ac.turnOn();
c.printOn();
System.out.println("----");

risc.turnOn();
sedile.turnOn();
c.printOn();
```

Output:

```
Aria condizionata
-----
Sedile riscaldato
Riscaldamento
```

309. (Pizza, 2014-11-3)

[CROWDGRADER] Realizzare la classe **Pizza**, in modo che ad ogni oggetto si possano assegnare degli ingredienti, scelti da un elenco fissato. Ad ogni ingrediente è associato il numero di calorie che apporta alla pizza. Gli oggetti di tipo **Pizza** sono dotati di ordinamento naturale, sulla base del numero totale di calorie. Infine, gli oggetti di tipo **Pizza** sono anche clonabili.

| | |
|--|---------|
| Esempio d'uso: | Output: |
| <pre>Pizza margherita = new Pizza(), marinara = new Pizza(); margherita.addIngrediente(Pizza.Ingrediente.POMODORO); margherita.addIngrediente(Pizza.Ingrediente.MOZZARELLA); marinara.addIngrediente(Pizza.Ingrediente.POMODORO); marinara.addIngrediente(Pizza.Ingrediente.AGLIO); Pizza altra = margherita.clone(); System.out.println(altra.compareTo(marinara));</pre> | 1 |

310. (2010-1-22)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A extends B {

    public A() {
        b1 = new B.C(true);
        b2 = new B(false);
    }

    public B f(Object o) {
        B x = super.f(o);
        return x.clone();
    }

    private B.C c = new B.C(3);
    private B b1, b2;
}
```

311. (2009-9-l'8)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {
    private A a = new B.C(3);
    private double x = a.f(3);
    private B b = new B.D(3);

    private int f(int n) {
        g(new B(3), n);
        return 2*n;
    }
    private void g(A u, int z) { }
    private void g(B u, double z) { }

    public A(int i) { }
}
```

312. (Washer, 2009-7-9)

La seguente classe rappresenta le operazioni elementari di una lavatrice:

```
class Washer {
    public void setTemp(int temp) { System.out.println("Setting_temperature_to_" + temp); }
    public void setSpeed(int rpm) { System.out.println("Setting_speed_to_" + rpm); }
    public void addSoap() { System.out.println("Adding_soap!"); }
}
```

Si implementi una classe **Program**, che rappresenta un programma di lavaggio per una lavatrice. Il metodo **addAction** aggiunge una nuova operazione elementare al programma. Un'operazione elementare può essere una delle tre operazioni elementari della lavatrice, oppure l'operazione

“Wait”, che aspetta un dato numero di minuti. Il metodo `execute` applica il programma ad una data lavatrice.

| | |
|---|--|
| Esempio d'uso: | Output dell'esempio d'uso: Setting temperature to 30 Setting speed to 20 Adding soap! (dopo 10 minuti) Setting speed to 100 Setting speed to 0 (dopo 10 minuti) |
| <pre> Washer w = new Washer(); Program p = new Program(); p.addAction(new Program.SetTemp(30)) ; p.addAction(new Program.SetSpeed(20)); p.addAction(new Program.Wait(10)); p.addAction(new Program.AddSoap()); p.addAction(new Program.SetSpeed (100)); p.addAction(new Program.Wait(10)); p.addAction(new Program.SetSpeed(0)); p.execute(w); </pre> | |

313. (2009-4-23)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    private B b1 = new B(null);
    private B b2 = new B.C();
    private B b3 = b1.new D();

    private int f(Object x) {
        if (x==null) throw b2;

        long l = b1.g();
        return b1.g();
    }
}

```

314. (2009-11-27)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```

public class A {
    public static final A a = new B(null);
    public final int n = B.f(3);

    public Object g() {
        B b = new B();
        B.C c = b.new C(7);
        return c;
    }

    public A(int i) { }
}

```

315. (Interval, 2009-1-29)

Si implementi la classe `Interval`, che rappresenta un intervallo di numeri reali. Un intervallo può essere chiuso oppure aperto, sia a sinistra che a destra. Il metodo `contains` prende come argomento un numero x e restituisce vero se e solo se x appartiene a questo intervallo. Il metodo `join` restituisce l'unione di due intervalli, senza modificarli, sollevando un'eccezione nel caso in cui questa unione non sia un intervallo. Si implementino anche le classi `Open` e `Closed`, in modo da rispettare il seguente caso d'uso.

| | |
|--|--|
| Esempio d'uso: <pre>Interval i1 = new Interval.Open(5, 10.5); Interval i2 = new Interval.Closed(7, 20); Interval i3 = i1.join(i2); System.out.println(i1.contains(5)); System.out.println(i1); System.out.println(i2); System.out.println(i3);</pre> | Output dell'esempio d'uso: false (5, 10.5) [7, 20] (5, 20] |
|--|--|

316. (Triangolo, 2008-4-21)

Nell'ambito di un programma di geometria, si implementi la classe **Triangolo**, il cui costruttore accetta le misure dei tre lati. Se tali misure non danno luogo ad un triangolo, il costruttore deve lanciare un'eccezione. Il metodo **getArea** restituisce l'area di questo triangolo. Si implementino anche la classe **Triangolo.Rettangolo**, il cui costruttore accetta le misure dei due cateti, e la classe **Triangolo.Iisoscele**, il cui costruttore accetta le misure della base e di uno degli altri lati.

Si ricordi che:

- Tre numeri a , b e c possono essere i lati di un triangolo a patto che $a < b + c$, $b < a + c$ e $c < a + b$.
- L'area di un triangolo di lati a , b e c è data da:

$$\sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)} \quad (\text{formula di Erone})$$

dove p è il semiperimetro.

| | |
|---|---|
| Esempio d'uso (fuori dalla classe Triangolo): <pre>Triangolo x = new Triangolo(10,20,25); Triangolo y = new Triangolo.Rettangolo(5,8); Triangolo z = new Triangolo.Iisoscele(6,5); System.out.println(x.getArea()); System.out.println(y.getArea()); System.out.println(z.getArea());</pre> | Output dell'esempio d'uso: 94.9918 19.9999 12.0 |
|---|---|

11 Classi enumerate

317. (LengthUnit, 2017-1-25)

Realizzare l'enumerazione `LengthUnit`, che rappresenta le principali unità di misura di lunghezza, dei sistemi metrico e imperiale: centimetri (CM), metri (M), chilometri (KM), pollici (INCH), iarde (YARD), e miglia (MILE). Il metodo `convertTo` accetta un'altra unità di misura u e un numero in virgola mobile x , e converte x da questa unità di misura a u .

I fattori di conversione per le misure imperiali sono i seguenti: 1 pollice = 0.025 metri, 1 iarda = 0.914 metri, 1 miglio = 1609 metri.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
System.out.println(LengthUnit.CM.convertTo(LengthUnit.INCH, 10));
System.out.println(LengthUnit.KM.convertTo(LengthUnit.YARD, 3.5));
);
System.out.println(LengthUnit.MILE.convertTo(LengthUnit.M, 6.2));
```

Output:

```
3.9370078740157486
3829.3216630196935
9975.8000000000001
```

318. (NutrInfo, 2014-7-3)

L'enumerazione `Nutrient` contiene i valori FAT, CARBO e PROTEIN. Implementare la classe `NutrInfo` che rappresenta la scheda nutrizionale di un prodotto gastronomico. Il suo costruttore accetta il numero di chilocalorie. Il metodo `setNutrient` permette di impostare la quantità (in grammi) di ciascun nutriente.

La classe è dotata di un ordinamento naturale, basato sul numero di calorie. Inoltre, la classe offre il metodo statico `comparatorBy` che accetta un valore `Nutrient` e restituisce un comparatore basato sul contenuto di quel nutriente.

Esempio d'uso:

```
NutrInfo x = new NutrInfo(500);
x.setNutrient(Nutrient.FAT, 12.0);
x.setNutrient(Nutrient.CARBO, 20.0);
x.setNutrient(Nutrient.PROTEIN, 15.0);
Comparator<NutrInfo> c = NutrInfo.comparatorBy(Nutrient.FAT);
```

319. (Status, 2014-3-5)

Implementare la classe enumerata `Status`, che rappresenta le 4 modalità di un programma di *instant messaging*: ONLINE, BUSY, HIDDEN e OFFLINE. Il metodo `isVisible` restituisce vero per i primi due e falso per gli altri. Il metodo `canContact` accetta un altro oggetto `Status` x e restituisce vero se un utente in questo stato può contattare un utente nello stato x e cioè se questo stato è diverso da OFFLINE e lo stato x è visibile.

Esempio d'uso:

```
Status a = Status.BUSY, b = Status.HIDDEN;
System.out.println(a.isVisible ());
System.out.println(a.canContact(b));
```

Output:

```
true
false
```

320. (Pizza, 2014-11-3)

[CROWDGRADER] Realizzare la classe `Pizza`, in modo che ad ogni oggetto si possano assegnare degli ingredienti, scelti da un elenco fissato. Ad ogni ingrediente è associato il numero di calorie che apporta alla pizza. Gli oggetti di tipo `Pizza` sono dotati di ordinamento naturale, sulla base del numero totale di calorie. Infine, gli oggetti di tipo `Pizza` sono anche clonabili.

| | |
|--|---------|
| Esempio d'uso: | Output: |
| <pre>Pizza margherita = new Pizza(), marinara = new Pizza(); margherita.addIngrediente(Pizza.Ingrediente.POMODORO); margherita.addIngrediente(Pizza.Ingrediente.MOZZARELLA); marinara.addIngrediente(Pizza.Ingrediente.POMODORO); marinara.addIngrediente(Pizza.Ingrediente.AGLIO); Pizza altra = margherita.clone(); System.out.println(altra.compareTo(marinara));</pre> | 1 |

321. (Coin, 2014-11-28)

Realizzare la classe enumerata Coin, che rappresenta le 8 monete dell'euro. Il metodo statico convert accetta un numero intero n e restituisce una collezione di Coin che vale n centesimi.

Nota: per ottenere lo stesso output del caso d'uso, non è necessario ridefinire alcun metodo `toString`.

| | |
|---|--|
| Esempio d'uso: | Output: |
| <pre>Collection<Coin> a = Coin.convert(34), b = Coin.convert(296); System.out.println(a); System.out.println(b);</pre> | [TWENTY, TEN, TWO, TWO] [TWOEUROS, FIFTY, TWENTY, TWENTY, FIVE, ONE] |

322. (BloodType, 2013-7-9)

Implementare l'enumerazione BloodType, che rappresenta i quattro gruppi sanguigni del sistema AB0, e cioè: A, B, AB e 0.

Il metodo canReceiveFrom accetta un altro oggetto BloodType x e restituisce true se questo gruppo sanguigno può ricevere trasfusioni dal gruppo x e false altrimenti.

Si ricordi che ogni gruppo può ricevere dal gruppo stesso e dal gruppo 0. In più, il gruppo AB può ricevere anche da A e da B (quindi, da tutti).

323. (Note, 2013-12-16)

Realizzare la classe enumerata Note, che rappresenta le sette note musicali. Le note sono disposte su una scala di frequenze, in modo che ciascuna nota sia separata dalla successiva da due semitonni, tranne il Mi, che è separato dal Fa da un solo semitono. La classe offre il metodo interval, che accetta un altro oggetto Note x e restituisce il numero di semitonni tra questa nota e la nota x .

Si faccia in modo che il metodo interval funzioni in tempo costante, cioè indipendente dall'argomento che gli viene passato.

| | |
|--|---------|
| Esempio d'uso: | Output: |
| <pre>Note a = Note.DO; System.out.println(a.interval(Note.MI)); System.out.println(Note.MI.interval(Note.LA)); System.out.println(Note.LA.interval(Note.SOL));</pre> | 4 5 -2 |

324. (NumberType, 2012-7-9)

Implementare la classe enumerata NumberType, che rappresenta i sei tipi primitivi numerici del linguaggio Java. Il campo width contiene l'ampiezza di questo tipo, in bit. Il metodo assignableTo prende come argomento un'altra istanza t di NumberType e restituisce vero se questo tipo è assegnabile a t (ovvero, c'è una conversione implicita dal tipo rappresentato da this al tipo rappresentato da t) e falso altrimenti.

| | |
|--|------------|
| Esempio d'uso: | Output: |
| <pre>System.out.println(NumberType.SHORT.width); System.out.println(NumberType.INT.isAssignableTo(NumberType.FLOAT));</pre> | 16 true |

325. (Panino, 2012-4-23)

Implementare la classe Panino, il cui metodo `addIngrediente` aggiunge un ingrediente, scelto da un elenco fisso. Gli ingredienti sono divisi in categorie. Se si tenta di aggiungere più di un ingrediente della stessa categoria, il metodo `addIngrediente` solleva un'eccezione.

Elenco delle categorie e degli ingredienti:

ripieni: PROSCIUTTO, SALAME

formaggi: SOTTILETTA, MOZZARELLA

salse: MAIONESE, SENAPE

| | |
|---|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Panino p = new Panino(); p.addIngrediente(Panino.Ingrediente.SALAME); p.addIngrediente(Panino.Ingrediente.SOTTILETTA); System.out.println(p); p.addIngrediente(Panino.Ingrediente.MOZZARELLA);</pre> | panino con SALAME, SOTTILETTA Exception in thread "main"... |

326. (TetrisPiece, 2010-7-26)

Implementare l'enumerazione Piece, che rappresenta i possibili pezzi del Tetris, con almeno le costanti T ed L, che rappresentano i pezzi dalla forma omonima. Il metodo `put` mette questo pezzo alle coordinate date di uno schema dato, con un dato orientamento. L'orientamento viene fornito dal chiamante tramite un intero compreso tra 0 (pezzo diritto) e 3 (pezzo ruotato di 270 gradi). Lo schema in cui sistemare il pezzo viene rappresentato da un array bidimensionale di valori booleani (`false` per libero, `true` per occupato). Il metodo `put` deve lanciare un'eccezione se non c'è posto per questo pezzo alle coordinate date.

Il seguente caso d'uso assume che `print_board` sia un opportuno metodo per stampare uno schema.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>boolean board[][] = new boolean[5][12]; Piece p1 = Piece.T; Piece p2 = Piece.L; p1.put(board, 0, 0, 0); p2.put(board, 0, 4, 0); p2.put(board, 1, 7, 2); print_board(board);</pre> | ----- X X XXX X XX XX X X |

327. (Cardinal, 2009-6-19)

Implementare l'enumerazione Cardinal, che rappresenta le 16 direzioni della rosa dei venti. Il metodo `isOpposite` prende come argomento un punto cardinale x e restituisce vero se questo punto cardinale è diametralmente opposto ad x , e falso altrimenti. Il metodo statico `mix` prende come argomento due punti cardinali, non opposti, e restituisce il punto cardinale intermedio tra i due.

| | |
|--|--|
| Esempio d'uso: | Output dell'esempio d'u- so: true NNE |
| Cardinal nord = Cardinal.N; System.out.println(nord.isOpposite(Cardinal.S)); Cardinal nordest = Cardinal.mix(Cardinal.N, Cardinal.E); assert nordest==Cardinal.NE : "Errore_inaspettato!"; Cardinal nordnordest = Cardinal.mix(nordest, Cardinal.N); System.out.println(nordnordest); | |

12 Vero o falso

328. (2022-3-28)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un'interfaccia funzionale può avere metodi default
- Il modificatore `volatile` offre garanzie di atomicità, visibilità e ordinamento
- `ArrayList<Integer>` è sottotipo di `List<? extends Number>`
- `Set<? extends Number>` è sottotipo di `Set<? super Number>`
- Una lambda espressione può estendere una classe esistente

329. (2020-2-27)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Secondo il pattern **Observer**, gli osservatori devono mantenere un riferimento al soggetto osservato.
- Il pattern **Composite** consente di aggregare oggetti in una struttura gerarchica.
- Il pattern **Strategy** consente ai clienti di fornire versioni nuove di un dato algoritmo.
- Il pattern **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato.
- In Java, una classe enumerata può avere costruttori multipli.
- In Java, `Set<Object>` è sottotipo di `Collection<?>`.

330. (2020-1-24)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Observer** vieta che lo stesso oggetto sia un osservatore di diversi soggetti.
- Nel pattern **Composite** un oggetto primitivo può contenere un altro oggetto primitivo.
- Il pattern **Decorator** serve ad aggiungere funzionalità a una classe senza modificarla.
- I metodi di accesso posizionale offerti da `List` rappresentano un'istanza del pattern **Iterator**.
- In Java, le classi enumerate possono avere campi (attributi) e metodi *custom*.
- In Java, `List<String>` è sottotipo di `List<Object>`.

331. (2019-9-20)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Le classi enumerate estendono implicitamente la classe `Enum`.
- La pre-condizione di un metodo è un obbligo a carico del chiamante.
- Secondo il pattern MVC per interfacce grafiche, i Controller interagiscono con i Model.

- Nel pattern **Decorator**, oggetto non decorato e oggetto decorato implementano la stessa interfaccia.
- Il pattern **Decorator** aggiunge funzionalità ad una classe senza modificarla.
- Nel pattern **Strategy** gli oggetti **Strategy** rappresentano varianti di un algoritmo.

332. (2019-7-23)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Comparator** è un'interfaccia funzionale.
- Un'interfaccia può contenere campi statici.
- Il pattern architettonale **MVC** per interfacce grafiche sfrutta il design pattern **Observer**.
- Il pattern **Decorator** prevede che il numero di decorazioni possibili sia illimitato.
- Il pattern **Decorator** prevede un metodo per rimuovere una decorazione da un oggetto decorato.
- Nel pattern **Strategy** la classe **Context** crea un oggetto che è sottotipo di **Strategy**.

333. (2019-6-24)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il modificatore **volatile** dà garanzie di atomicità, visibilità, e ordinamento.
- Per poter invocare `x.join()` è necessario possedere il monitor dell'oggetto `x`.
- Se `f` e `g` sono due metodi **synchronized**, le chiamate `x.f()` e `x.g()` sono mutuamente esclusive.
- Il pattern **Decorator** consente di applicare una decorazione a un oggetto già decorato.
- Il pattern **Decorator** prevede un metodo per rimuovere una decorazione da un oggetto decorato.
- Il pattern **Strategy** può essere implementato tramite un'enumerazione che elenca le possibili varianti dell'algoritmo.

334. (2019-3-19)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Observer**, più oggetti possono osservare lo stesso oggetto.
- Nel pattern **Observer**, gli osservatori devono mantenere un riferimento all'oggetto osservato.
- Il pattern **Composite** consente a un oggetto composito di contenere altri oggetti compositi.
- Il pattern **Decorator** è un modo di aggiungere funzionalità a un oggetto a runtime.
- Una classe enumerata può avere campi privati.

335. (2019-2-15)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Strategy** prevede che tra le varianti di un algoritmo ce ne sia una di default.

- Il pattern **Decorator** vieta di aggiungere una decorazione ad un oggetto già decorato.
- Il pattern **Factory Method** consente a diverse sottoclassi di creare prodotti concreti diversi.
- Nel pattern **Factory Method** il prodotto generico è sottoclasse del produttore generico.
- Una classe enumerata può implementare un'interfaccia.

336. (2019-10-9)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Una classe enumerata può implementare un'interfaccia.
- Una lambda espressione può non avere argomenti.
- Un'interfaccia funzionale può avere diversi metodi default, ma deve avere un unico metodo astratto.
- La scelta del layout di un componente Swing/AWT è un esempio del pattern **Decorator**.
- Il pattern **Decorator** permette di aggiungere a una classe quelle funzionalità che non hanno bisogno di nuovi metodi.
- Nel pattern **Strategy** gli oggetti **Strategy** rappresentano varianti di un algoritmo.

337. (2019-1-23)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Una classe enumerata può avere più di un costruttore
- Il pattern **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato
- Il contesto del pattern **Strategy** prevede che esista un numero predefinito di varianti di un algoritmo
- Il contesto del pattern **Decorator** prevede che un oggetto decorato si possa utilizzare come uno non decorato
- I pattern **Composite** e **Decorator** sono soluzioni alternative allo stesso problema

338. (2018-9-17)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il modificatore **volatile** dà garanzie di atomicità e visibilità, ma non di ordinamento.
- Aggiungere un tasto (**JButton**) ad una finestra AWT rappresenta un'applicazione del pattern **Decorator**.
- Il pattern **Strategy** si implementa tipicamente utilizzando una classe enumerata.
- Il pattern **Composite** prevede che un contenitore si possa comportare come un oggetto primitivo.
- Il pattern **Observer** consente che lo stesso oggetto sia un osservatore di diversi soggetti.

339. (2018-7-19)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Observer** vieta che lo stesso oggetto sia un osservatore di diversi soggetti.
- Nel pattern **Composite** oggetto primitivo e oggetto composito implementano una stessa interfaccia.
- Il pattern **Decorator** si applica anche quando il numero di decorazioni possibili è limitato.
- I metodi di accesso posizionale offerti da **List** rappresentano un'istanza del pattern **Iterator**.
- Il pattern **Strategy** permette ai clienti di fornire una variante di un algoritmo.

340. (2018-6-20)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Observer** vieta che lo stesso oggetto sia un osservatore di diversi soggetti.
- Il pattern **Observer** prevede che osservatore e soggetto osservato implementino una stessa interfaccia.
- Il pattern **Decorator** consente di aggiungere funzionalità a una classe senza modificarla.
- I metodi di accesso posizionale offerti da **List** rappresentano un'istanza del pattern **Iterator**.
- I design pattern offrono una casistica dei più comuni errori di progettazione.

341. (2018-3-23)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Composite** e il pattern **Decorator** sono soluzioni alternative allo stesso problema.
- Uno dei pre-requisiti del pattern **Strategy** è che non esista un numero predefinito di varianti dell'algoritmo.
- Il pattern **Composite** prevede che gli oggetti contenuti conoscano il contenitore in cui sono stati inseriti.
- Nel pattern **Decorator**, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- Il modificatore **volatile** si può applicare a una variabile locale.
- Un thread può rimanere in attesa di entrare in un blocco **synchronized** a tempo indefinito.
- Ad ogni thread di esecuzione è associato un oggetto della classe **Thread**.
- La classe **Thread** offre un metodo per aspettare la terminazione di un altro thread.

342. (2018-2-22)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Factory Method** il produttore generico ha una relazione di dipendenza dal prodotto generico.
- L'aggiunta di un pulsante (**JButton**) ad un pannello **AWT** rappresenta un'applicazione del pattern **Strategy**.
- Il pattern **Decorator** consente di aggiungere funzionalità ad una classe senza modificarla.

- Il pattern **Strategy** serve a migliorare l'efficienza di un algoritmo.
- Il pattern **Observer** suggerisce di organizzare gli osservatori in una gerarchia di classi e sotto-classi.

343. (2018-10-18)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il modificatore **volatile** dà garanzie di atomicità, visibilità, e ordinamento.
- Per poter invocare `x.notify()` è necessario possedere il monitor dell'oggetto `x`.
- Tipicamente, nel pattern **Decorator**, per decorare un oggetto lo si passa al costruttore di un decoratore.
- Nel pattern **Decorator**, i decoratori sono sottoclassi dell'oggetto base non decorato (**ConcreteComponent**).
- Il pattern **Composite** richiede che ci sia un metodo per rimuovere un oggetto da un contenitore.
- Il pattern **Observer** prevede che il soggetto generatore di eventi conservi un riferimento ai suoi osservatori.

344. (2018-1-24)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Factory Method** il prodotto concreto è sottotipo del prodotto generico.
- Aggiungere un tasto (`JButton`) ad una finestra **AWT** rappresenta un'applicazione del pattern **Factory Method**.
- Il pattern **Composite** prevede che un contenitore si possa comportare come un oggetto primitivo.
- Il pattern **Decorator** consente di aggiungere una ulteriore decorazione ad un oggetto già decorato.

345. (2017-7-20)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Observer** anche il soggetto osservato implementa l'interfaccia **Observer**
- La scelta del layout di un componente Swing/AWT rappresenta un'istanza del pattern **Strategy**
- Una delle precondizioni del pattern **Factory Method** è che vi siano più tipi di prodotti concreti.
- Il metodo `add` dell'interfaccia **Collection** rappresenta un'istanza del pattern **Strategy**.
- Nel pattern **Decorator**, tipicamente un oggetto viene decorato passandolo al costruttore di una classe decoratrice.

346. (2017-6-21)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Observer** gli osservatori interrogano periodicamente il soggetto che genera eventi.

- Il pattern **Strategy** suggerisce di usare oggetti per rappresentare varianti di un algoritmo.
- Tutte le classi astratte sono applicazioni del pattern **Template Method**.
- Il metodo **add** dell'interfaccia **Collection** rappresenta un'istanza del pattern **Factory Method**.
- Il pattern **Composite** prevede che si possa iterare sui componenti di un oggetto composto.
- Una variabile locale può essere **volatile**.

347. (2017-3-23)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Decorator**, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- Il pattern **Composite** prevede che il tipo effettivo degli oggetti contenuti (**Leaf**) sia sottotipo del tipo effettivo dell'oggetto contenitore (**Composite**).
- I design pattern offrono una casistica dei più comuni errori di progettazione.
- Il pattern **Composite** e il pattern **Decorator** sono soluzioni alternative allo stesso problema.
- Una delle premesse del pattern **Strategy** è che un algoritmo abbia un numero prefissato di varianti.
- Una classe interna statica non può essere istanziata se non viene prima istanziata la classe contenitrice.

348. (2017-2-23)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Decorator**, l'oggetto decorato ha dei riferimenti agli oggetti decoratori.
- Il metodo **toString** della classe **Object** rappresenta un esempio del pattern **Factory Method**.
- Il pattern **Composite** prevede che gli oggetti primitivi abbiano un riferimento al contenitore in cui sono inseriti.
- La scelta del layout di un container **AWT** rappresenta un'istanza del pattern **Strategy**.
- Il pattern **Observer** prevede un'interfaccia che sarà implementata da tutti gli osservatori.
- Una classe astratta può avere un costruttore.

349. (2017-10-6)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato.
- Nel pattern **Factory Method** il prodotto generico è sottotipo del produttore generico.
- Uno dei pre-requisiti del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.
- Il pattern **Composite** prevede che gli oggetti contenuti conoscano il contenitore in cui sono stati inseriti.

- Il modificatore `volatile` si può applicare anche a un intero metodo
- La classe `LinkedList<T>` è sottotipo di `ArrayList<T>`
- La classe `LinkedList<T>` è sottotipo di `List<? extends T>`
- La classe `LinkedList<T>` è sottotipo di `List<? super T>`
- L'espressione `null instanceof String` ha valore `true`

350. (2017-1-25)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Observer**, il soggetto osservato avvisa gli osservatori degli eventi rilevanti.
- Una delle premesse del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.
- Nel pattern **Composite**, un oggetto composito può essere vuoto.
- Nel pattern **Factory Method**, il prodotto generico è sottotipo del produttore generico.
- Il pattern **Decorator** consente di aggiungere funzionalità a una classe senza modificarla.
- Una variabile `volatile` deve essere di tipo primitivo.
- L'interfaccia `Map<K,V>` estende `Collection<K>`.
- Per ogni oggetto `x`, dovrebbe valere `x.equals(x)==true`.
- Le implementazioni di `hashCode` e `equals` nella classe `Object` sono coerenti.
- Una volta clonato, un oggetto non dovrebbe più essere modificato.

351. (2016-9-20)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Observer** gli osservatori interrogano periodicamente il soggetto osservato.
- Uno dei pre-requisiti del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.
- Il pattern **Factory Method** si applica quando un oggetto deve contenere altri.
- Il pattern **Composite** impedisce di distinguere un oggetto primitivo da uno composito.
- Nel pattern **Decorator**, per “decorare” si intende “aggiungere funzionalità”.
- Una variabile `volatile` deve essere di tipo primitivo.

352. (2016-7-21)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Decorator**, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- Aggiungere un campo di testo (`JTextView`) ad una finestra AWT rappresenta un'applicazione del pattern **Decorator**.
- Nel pattern **Strategy**, la classe `Context` ha un metodo che restituisce un oggetto di tipo `Strategy`.
- Una delle premesse del pattern **Factory Method** è che i produttori creino prodotti di tipo diverso.
- Il modificatore `volatile` si può applicare a campi e metodi.
- Il Java Memory Model offre garanzie di performance.

353. (2016-6-22)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Observer**, il soggetto generatore di eventi ha la responsabilità di notificare gli osservatori.
- Aggiungere un tasto (**JButton**) ad una finestra **AWT** rappresenta un'applicazione del pattern **Decorator**.
- Nel pattern **Composite**, contenitori e oggetti primitivi implementano la stessa interfaccia.
- Nel pattern **Strategy** un oggetto rappresenta una versione di un algoritmo.
- Il pattern **Iterator** si applica ogni volta si debba svolgere la stessa operazione ripetutamente.

354. (2016-3-3)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Factory Method** si applica quando un oggetto deve contenerne altri.
- Nel pattern **Strategy** la classe **Context** ha un metodo che accetta un oggetto di tipo **Strategy**.
- Nel pattern **Decorator** gli oggetti primitivi posseggono un metodo per aggiungere una decorazione.
- Il metodo **add** dell'interfaccia **Collection** rappresenta un'istanza del pattern **Template Method**.
- Composite** e **Decorator** hanno diagrammi UML simili, tranne che per la molteplicità di una aggregazione.

355. (2016-1-27)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Factory Method** prevede un prodotto generico e vari prodotti specifici.
- Il pattern **Decorator** prevede che oggetti primitivi e decoratori implementino una stessa interfaccia.
- Il metodo **Collections.sort** rappresenta un'istanza del pattern **Template Method**.
- Secondo il pattern **Observer**, l'oggetto osservato conserva riferimenti ai suoi osservatori.
- Il pattern **Strategy** permette di fornire versioni diverse di un algoritmo.

356. (2015-9-21)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Decorator**, l'oggetto decorato conserva dei riferimenti agli oggetti decoratori.
- Il pattern **Strategy** e il pattern **Observer** sono soluzioni alternative allo stesso problema.
- Nel pattern **Composite**, un oggetto composito ne può contenere un altro.
- Il pattern **Iterator** consente di esaminare il contenuto di una collezione senza esporre la sua struttura interna.

- I design pattern sono soluzioni consigliate per problemi ricorrenti di programmazione.

357. (2015-7-8)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern Observer, l'osservatore ha un metodo per registrarsi presso un oggetto da osservare.
- Aggiungere un tasto (JButton) ad una finestra AWT rappresenta un'applicazione del pattern Decorator.
- Il pattern Composite prevede che un contenitore si possa comportare come un oggetto primitivo.
- Il pattern Factory Method prevede che una classe costruisca oggetti di un'altra classe.
- Il pattern Strategy si implementa tipicamente con una classe astratta.

358. (2015-6-24)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern Decorator, l'oggetto decorato è sotto-tipo dell'oggetto decoratore.
- Il pattern Iterator consente di esaminare una collezione senza esporre la sua struttura interna.
- L'interfaccia Comparator rappresenta un'istanza del pattern Template Method.
- Lo scopo del pattern Composite è di aggiungere funzionalità ad una data classe.
- Nel pattern Factory Method, i prodotti concreti sono sotto-tipi del prodotto generico.

359. (2015-2-5)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Tipicamente l'aggregazione tra decoratore e oggetto decorato viene stabilita da un costruttore.
- Il pattern Strategy consente ai clienti di fornire versioni particolari di un algoritmo.
- Le interfacce Iterator e Iterable rappresentano un'istanza del pattern Factory Method.
- Il pattern Template Method prevede che un metodo concreto di una classe ne invochi uno astratto della stessa classe.
- L'interfaccia Collection è un'istanza del pattern Composite.

360. (2015-1-20)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- In Java ad ogni thread di esecuzione è sempre associato un oggetto Thread.
- Un thread non può invocare interrupt su sé stesso (cioè, sull'oggetto Thread che gli corrisponde).
- wait è un metodo di Thread.
- Invocare x.wait() senza possedere il mutex di x provoca un errore di compilazione.
- Un campo di classe non può essere synchronized.

- Il pattern **Composite** prevede che gli oggetti primitivi abbiano un riferimento al contenitore in cui sono inseriti.
- Nel framework **MVC**, ogni oggetto *view* comunica con almeno un oggetto *model*.
- La scelta del layout di un container **AWT** rappresenta un'istanza del pattern **Composite**.
- Nel pattern **Factory Method** i client non hanno bisogno di conoscere il tipo effettivo dei prodotti.
- Il pattern **Decorator** prevede che l'oggetto da decorare abbia un metodo per aggiungere una decorazione.

361. (2014-9-18)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il metodo **hashCode** di **Object** rappresenta un'istanza del pattern **Template Method**.
- Il pattern **Composite** prevede che si possa iterare sui componenti di un oggetto composto.
- Il pattern **Factory Method** consente a diverse sottoclassi di creare prodotti diversi.
- Nel pattern **Factory Method** il prodotto generico è sottotipo del produttore generico.
- Uno dei pre-requisiti del pattern **Iterator** è che più client debbano poter accedere contemporaneamente all'aggregato.

362. (2014-7-3)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Observer** evita che gli osservatori debbano controllare periodicamente lo stato dell'oggetto osservato (*polling*).
- Il pattern **Decorator** prevede un modo per distinguere un oggetto decorato da uno non decorato.
- Il pattern **Template Method** si applica in presenza di una gerarchia di classi e sottoclassi.
- L'interfaccia **Collection** è un'istanza del pattern **Composite**.
- L'interfaccia **Collection** sfrutta il pattern **Iterator**.

363. (2014-7-28)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern **Composite** sia gli oggetti primitivi che quelli compositi hanno un metodo per aggiungere un oggetto alla composizione.
- Il metodo **clone** di **Object** è un'istanza del pattern **Template Method**.
- Il pattern **Factory Method** consente a diverse sottoclassi di creare prodotti diversi.
- Le interfacce **Iterable** e **Iterator** rappresentano un'istanza del pattern **Factory Method**.
- Il pattern **Strategy** prevede un'interfaccia (o classe astratta) che rappresenta un algoritmo.

364. (2014-3-5)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Iterator** consente ad una collezione di scorrere i propri elementi senza esporre la sua struttura interna.
- Nel pattern **Observer** l'oggetto osservato conserva dei riferimenti ai suoi osservatori.
- Il pattern **Strategy** usa un oggetto per rappresentare una variante di un algoritmo.
- Passare un **Comparator** al metodo `Collections.sort` rappresenta un'istanza del pattern **Strategy**.
- Il metodo `hashCode` di `Object` rappresenta un'istanza del pattern **Template Method**.

365. (2014-11-28)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Template Method** consente alle sottoclassi di definire versioni concrete di metodi primitivi.
- Il pattern **Decorator** consente anche di aggiungere una decorazione ad un oggetto già decorato.
- Il pattern **Factory Method** consente a diverse sottoclassi di creare prodotti diversi.
- Nel pattern **Factory Method** i produttori concreti sono sottoclassi del produttore generico.
- Uno dei pre-requisiti del pattern **Strategy** è che esista un numero predefinito di varianti di un algoritmo.

366. (2014-1-31)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Iterator** prevede un metodo che contemporaneamente restituisce il prossimo oggetto e fa avanzare di un posto l'iteratore.
- Nel pattern **Decorator** l'oggetto da decorare ha un metodo per aggiungere una decorazione.
- Ogni qual volta un metodo accetta come argomento un altro oggetto siamo in presenza del pattern **Strategy**.
- Il pattern **Factory Method** prevede che un oggetto ne crei un altro.
- Il pattern **Template Method** può essere implementato utilizzando classi astratte.

367. (2013-9-25)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Composite** consente di aggregare gli oggetti in una struttura ad albero.
- In Java, il pattern **Composite** prevede che oggetti primitivi e compositi implementino la stessa interfaccia.
- Nel pattern **Decorator**, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- Il pattern **Observer** può essere utilizzato per notificare gli eventi generati da un'interfaccia grafica.
- Il metodo `hashCode` di `Object` rappresenta un'istanza del pattern **Template Method**.

368. (2013-7-9)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Composite** prevede che si possa iterare sui componenti di un oggetto composto.
- Nel pattern **Decorator**, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- Nel pattern **Observer**, il soggetto osservato mantiene riferimenti a tutti gli osservatori.
- Nel pattern **Observer**, gli osservatori hanno un metodo per registrare un soggetto da osservare.
- La scelta del layout di un container **AWT** rappresenta un'istanza del pattern **Strategy**.

369. (2013-6-25)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Iterator** prevede un metodo per far ripartire l'iteratore daccapo.
- Il pattern **Observer** evita che gli osservatori debbano controllare periodicamente lo stato dell'oggetto osservato (*polling*).
- Di norma, il pattern **Decorator** si applica solo quando l'insieme delle decorazioni possibili è illimitato.
- Il pattern **Composite** consente anche agli oggetti primitivi di contenere altri.
- In Java, il pattern **Template Method** viene comunemente implementato usando una classe astratta.

370. (2013-3-22)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Observer** permette a diversi oggetti di ricevere gli eventi generati dallo stesso oggetto.
- Il pattern **Observer** prevede che osservatore e soggetto osservato implementino una stessa interfaccia.
- Il pattern **Composite** consente la composizione ricorsiva di oggetti.
- Il pattern **Iterator** permette a diversi thread di iterare contemporaneamente sulla stessa collezione.
- I design pattern offrono una casistica dei più comuni errori di progettazione.

371. (2013-2-11)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Il pattern **Composite** prevede che gli oggetti primitivi abbiano un riferimento al contenitore in cui sono inseriti.
- Il pattern **Composite** prevede che un contenitore si possa comportare come un oggetto primitivo.
- Nel framework **MVC**, ogni oggetto *view* comunica con almeno un oggetto *model*.
- La scelta del layout di un container **AWT** rappresenta un'istanza del pattern **Strategy**.

- Il pattern Observer prevede un'interfaccia che sarà implementata da tutti gli osservatori.

372. (2013-12-16)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- L'annidamento di componenti grafici Swing/AWT sfrutta il pattern Composite.
- Tipicamente, nel pattern Decorator l'oggetto da decorare viene passato al costruttore dell'oggetto decoratore.
- Il pattern Template Method si applica in presenza di una gerarchia di classi e sottoclassi.
- Il pattern Factory Method si applica quando un oggetto deve contenere altri.
- Nel pattern Strategy la classe Context ha un metodo che accetta un oggetto di tipo Strategy.

373. (2013-1-22)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Nel pattern Decorator, l'oggetto decoratore si comporta come l'oggetto da decorare.
- Nel pattern Decorator, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- Il metodo Collections.sort rappresenta un'istanza del pattern Template Method.
- Secondo il pattern Observer, gli osservatori devono contenere un riferimento all'oggetto osservato.
- Il pattern Strategy permette di fornire versioni diverse di un algoritmo.

374. (2012-9-3)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- L'operatore .class si applica ad un riferimento
- Tramite riflessione è possibile invocare un metodo il cui nome è sconosciuto a tempo di compilazione
- ArrayList<Integer> è sottotipo di Set<?>
- ArrayList<Integer> è sottotipo di List<? extends Number>
- L'eccezione ArrayIndexOutOfBoundsException è verificata (*checked*)
- Nel pattern Observer, il soggetto osservato ha dei riferimenti ai suoi osservatori.
- La ridefinizione (*overriding*) del metodo equals da parte di una classe rappresenta un esempio del pattern Strategy.
- Nel pattern Decorator, l'oggetto decorato ha dei riferimenti agli oggetti decoratori.
- Il metodo `toString` della classe Object rappresenta un esempio del pattern Factory Method
- Il pattern Strategy e il pattern Template Method sono soluzioni alternative allo stesso problema.

375. (2012-7-9)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Una classe astratta può avere metodi concreti (dotati di corpo).
- Una classe interna può essere **protected**.
- ArrayIndexOutOfBoundsException** è una eccezione verificata (*checked*).
- RandomAccess** è una interfaccia parametrica.
- La classe **Thread** ha un costruttore senza argomenti.
- Nel pattern **Observer**, più oggetti possono osservare lo stesso oggetto.
- Il pattern **Observer** prevede un'interfaccia che sarà implementata da tutti gli osservatori.
- Il pattern **Template Method** suggerisce l'utilizzo di una classe astratta.
- Il pattern **Composite** organizza degli oggetti in una gerarchia ad albero.
- Nel framework **MVC**, le classi **Controller** si occupano dell'interazione con l'utente.

376. (2012-6-18)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Le enumerazioni estendono implicitamente la classe **Enum**.
- Comparable<Integer>** è sottotipo di **Comparable<Number>**.
- Integer** è sottotipo di **Number**.
- Cloneable** è una interfaccia vuota.
- Due oggetti con lo stesso *hash code* dovrebbero essere considerati uguali da *equals*.
- Il pattern **Strategy** suggerisce di utilizzare un oggetto per rappresentare un algoritmo.
- Il modo in cui si associa un gestore di eventi alla pressione di un JButton in Swing/AWT rappresenta un'istanza del pattern **Strategy**.
- Il pattern **Iterator** prevede un metodo per far ripartire l'iteratore daccapo.
- Il pattern **Composite** prevede un metodo per distinguere un oggetto primitivo da uno composito.
- Il pattern **Decorator** prevede che si possa utilizzare un oggetto decorato nello stesso modo di uno non decorato.

377. (2011-3-4)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Iterable<?>** è supertipo di **Iterator<Integer>**.
- Runnable** è un'interfaccia.
- clone** è un metodo pubblico di **Object**.
- Le variabili locali possono essere **final**.
- wait** è un metodo di **Thread**.
- Una classe interna statica può avere campi istanza (attributi).
- Nel pattern **Decorator**, l'oggetto decorato mantiene un riferimento a quello decoratore.
- Nel pattern **Template Method**, una classe ha un metodo concreto che chiama metodi astratti.
- Nel pattern **Factory Method**, il prodotto generico dovrebbe essere sottotipo del produttore generico.

- La scelta del layout di un componente grafico è un esempio di applicazione del pattern Template Method.

378. (2011-2-7)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Collection<?> è supertipo di Set<Integer>.
- TreeMap è un'interfaccia.
- Tutte le classi implementano automaticamente Cloneable.
- getClass è un metodo pubblico di Object.
- Enum è una classe parametrica.
- Di un metodo final non si può fare l'overloading.
- Nel pattern Decorator, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- Le interfacce Iterator e Iterable sono un esempio del pattern Template Method.
- Nel framework Model-View-Controller, gli oggetti Model sono indipendenti dall'interfaccia utente utilizzata.
- Il pattern Composite prevede che il tipo effettivo degli oggetti contenuti sia sottotipo del tipo effettivo dell'oggetto contenitore.

379. (2010-9-14)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un campo statico è condiviso da tutti gli oggetti della sua classe.
- RandomAccess è una interfaccia parametrica.
- Gli oggetti di tipo Integer sono immutabili.
- Le enumerazioni sono sempre clonabili.
- Un file sorgente Java può contenere più classi.
- I design pattern offrono una casistica dei più comuni errori di progettazione.
- Il pattern Strategy suggerisce di utilizzare un oggetto per rappresentare un algoritmo.
- Il metodo `toString` della classe Object rappresenta un esempio del pattern Factory Method.
- Il pattern Observer si applica quando un oggetto genera eventi destinati ad altri oggetti.
- Il pattern Composite e il pattern Decorator sono soluzioni alternative allo stesso problema.

380. (2010-7-26)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- interrupt è un metodo della classe Object.
- Iterator<?> è supertipo di Iterator<? extends Employee>.
- Runnable è una interfaccia vuota.
- Un metodo statico può essere sincronizzato (`synchronized`).
- Il metodo `notify` risveglia uno dei thread in attesa su questo oggetto.

12 Vero o falso

- Il metodo `notify` può lanciare l'eccezione `InterruptedException`.
- Nel pattern `Decorator`, ogni oggetto decoratore contiene un riferimento all'oggetto decorato.
- Il pattern `Iterator` consente ad un oggetto di contenerne altri.
- Il pattern `Composite` consente di aggiungere funzionalità ad una classe.
- Nel pattern `Factory Method`, i produttori concreti sono tutti sotto-tipi del produttore generico.

381. (2010-6-28)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- L'interfaccia `Map<K,V>` estende `Iterable<K>`.
- `Iterator<?>` è supertipo di `Iterator<? extends Employee>`.
- `RandomAccess` è una interfaccia vuota.
- Una classe astratta può estendere un'altra classe astratta.
- Una classe interna può avere visibilità `protected`.
- Una classe anonima non può avere costruttore.
- Nel pattern `Decorator`, non è necessario che l'oggetto decorato sia consapevole della decorazione.
- Nel framework MVC, le classi "model" si occupano di presentare i dati all'utente.
- Il pattern `Strategy` si applica quando un algoritmo si basa su determinate operazioni primitive.
- Il pattern `Factory Method` permette ad una gerarchia di produttori di produrre una gerarchia di prodotti.

382. (2010-2-24)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- E' possibile sincronizzare un blocco di codice rispetto a qualsiasi oggetto.
- Il metodo `wait` mette in attesa il thread corrente.
- Una classe enumerata (`enum`) può implementare un'interfaccia.
- Una classe astratta può estendere un'altra classe astratta.
- Di un metodo `final` non è possibile fare l'overloading.
- Un costruttore non può lanciare eccezioni.
- Nel pattern `Observer`, il soggetto osservato avvisa gli osservatori degli eventi rilevanti.
- Nel framework MVC, le classi view si occupano di presentare i dati all'utente.
- Il pattern `Template Method` si applica quando un algoritmo si basa su determinate operazioni primitive.
- Il pattern `Factory Method` si applica quando una classe deve costruire oggetti di un'altra classe.

383. (2010-11-30)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- L'interfaccia `List<E>` estende `Iterator<E>`.

- Collection<?> è supertipo di Set<Integer>.
- Un metodo statico può essere astratto.
- notify è un metodo pubblico di Object.
- start è un metodo statico di Thread.
- L'invocazione x.join() mette il thread corrente in attesa che il thread x termini.
- Nel pattern Decorator, l'oggetto decorato e quello decoratore implementano una stessa interfaccia.
- Le interfacce Iterator e Iterable sono un esempio del pattern Composite.
- Il pattern Template Method si applica quando un algoritmo si basa su determinate operazioni primitive.
- Il pattern Observer permette a diversi oggetti di ricevere gli eventi generati da un altro oggetto.

384. (2010-1-22)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- La classe Object ha un metodo sleep.
- Il metodo wait può lanciare un'eccezione verificata.
- Il metodo notify sveglia tutti i thread in attesa su quest'oggetto.
- Un metodo statico non può essere synchronized.
- La classe HashMap<K,V> estende l'interfaccia Collection<K>.
- Una classe enumerata (enum) può estenderne un'altra.
- Nel pattern Observer, ogni osservatore conosce tutti gli altri.
- La scelta del layout di una finestra in AWT rappresenta un'applicazione del pattern Strategy.
- Il pattern Composite permette di trattare un insieme di elementi come un elemento primitivo.
- Il pattern Decorator si applica quando l'insieme delle decorazioni possibili è illimitato.

385. (2009-9-1'8)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- equals è un metodo final della classe Object.
- Una classe può implementare più interfacce contemporaneamente.
- Il metodo add di Set funziona sempre in tempo costante.
- Il metodo add di List non restituisce mai false.
- Il modificatore transient indica che quel campo non deve essere serializzato.
- Le classi enumerate non possono essere istanziate con new.
- Il pattern Decorator permette di aggiungere funzionalità ad una classe.
- Inserire una serie di oggetti in una LinkedList rappresenta un'istanza del pattern Composite.
- Il pattern Factory Method suggerisce che il "prodotto generico" sia sottotipo del "produttore generico".
- Ridefinire il metodo clone tramite overriding rappresenta un'istanza del pattern Strategy.

386. (2009-7-9)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- `equals` è un metodo statico della classe Object.
- L'interfaccia Cloneable è vuota.
- Uno dei metodi sort di Collections prende come argomento una Collection.
- Il metodo getClass di Object restituisce la classe effettiva di this.
- Qualunque oggetto può essere lanciato con throw.
- HashSet<Integer> è sottotipo di Iterable<Integer>.
- Il pattern Decorator si applica quando c'è un insieme prefissato di decorazioni possibili.
- Il pattern Iterator prevede un metodo per rimuovere l'ultimo elemento visitato dall'iteratore.
- Il metodo `toString` di Object rappresenta un'istanza del pattern Factory Method.
- Ridefinire il metodo `equals` tramite overriding rappresenta un'istanza del pattern Strategy.

387. (2009-6-19)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- `sleep` è un metodo statico della classe Object.
- Il metodo `wait` di Object prende un argomento.
- Il metodo `clone` di Object effettua una copia superficiale.
- L'interfaccia Serializable è vuota.
- Un'eccezione verificata non può essere catturata.
- List<Integer> è sottotipo di Iterable<Integer>.
- Il pattern Template Method si può applicare quando un algoritmo utilizza delle *operazioni primitive*.
- Nel framework MVC, le classi controller gestiscono gli eventi dovuti all'interazione con l'utente.
- Il metodo `iterator` dell'interfaccia Iterable rappresenta un'istanza del pattern Factory Method.
- L'aggiunta di un Component AWT dentro un altro Component rappresenta un'istanza del pattern Decorator.

388. (2009-2-19)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un'interfaccia può essere dichiarata final.
- Una classe interna può essere dichiarata final.
- Il metodo `equals` di Object effettua un confronto di indirizzi.
- LinkedList<Integer> è assegnabile a LinkedList<?>.
- E' possibile lanciare con throw qualunque oggetto.
- La relazione di "sottotipo" è una relazione di equivalenza tra classi (è riflessiva, simmetrica e transitiva).

- Il pattern **Template Method** si può applicare quando un algoritmo utilizza delle *operazioni primitive*.
- Nel framework **MVC**, le classi view si occupano di presentare il modello all'utente.
- Nel pattern **Strategy**, si definisce un'interfaccia che rappresenta un algoritmo.
- Il pattern **Iterator** si applica ad un aggregato che non vuole esporre la sua struttura interna.

389. (2009-11-27)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- La classe **Thread** ha un costruttore che accetta un **Runnable**.
- Runnable** è un'interfaccia vuota.
- La serializzazione è un modo standard di memorizzare oggetti su file.
- Se due oggetti sono uguali secondo il metodo **equals**, dovrebbero avere lo stesso codice hash secondo il metodo **hashCode**.
- L'interfaccia **Comparator<T>** estende l'interfaccia **Comparable<T>**.
- Le classi enumerate non possono essere istanziate con **new**.
- Nel pattern **Observer**, più oggetti possono osservare lo stesso oggetto.
- Il pattern **Composite** prevede un'interfaccia che rappresenti un oggetto primitivo.
- Le interfacce **Iterator** e **Iterable** rappresentano un'istanza del pattern **Factory Method**.
- Ridefinire il metodo **clone** tramite overriding rappresenta un'istanza del pattern **Template Method**.

390. (2009-1-29)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- La classe **Thread** è astratta.
- Il metodo **clone** di **Object** è pubblico.
- Il metodo **clone** di **Object** effettua una copia superficiale.
- Un campo (o attributo) statico può essere **private**.
- Un'eccezione non-verificata non può essere catturata.
- getClass** è un metodo della classe **Object**.
- Il pattern **Strategy** si può applicare quando un algoritmo utilizza delle *operazioni primitive*.
- Nel framework **MVC**, le classi view ricevono gli eventi dovuti all'interazione con l'utente.
- Nel pattern **Decorator**, l'oggetto decoratore maschera (cioè, fa le veci del) l'oggetto decorato.
- L'aggiunta di un **Component** AWT dentro un altro **Component** rappresenta un'istanza del pattern **Composite**.

391. (2009-1-15)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un campo statico viene inizializzato al caricamento della classe.

- Object è assegnabile a String.
- Iterable è un'interfaccia parametrica.
- Una classe astratta può estenderne un'altra.
- Si può scrivere “public interface I<Integer> { }”.
- Un metodo statico può essere astratto.
- La scelta del layout di un container AWT rappresenta un’istanza del pattern Template Method.
- Iterator<T> estende Iterable<T>.
- Nel framework MVC, ogni oggetto *view* comunica con almeno un oggetto *model*.
- Il pattern Observer prevede un’interfaccia che sarà implementata da tutti gli osservatori.

392. (2008-9-8)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un Double occupa più spazio in memoria di un double.
- int è sottotipo di long.
- Runnable è assegnabile a Object.
- Una classe anonima può avere un costruttore.
- instanceof opera sul tipo effettivo del suo primo argomento.
- Una classe interna statica non può avere metodi di istanza.
- Il pattern Iterator prevede un metodo per far ripartire l'iteratore daccapo.
- Nel framework MVC, le classi View si occupano della presentazione dei dati all'utente.
- La classe Java ActionListener rappresenta un'applicazione del pattern Observer.
- L'aggiunta di un tasto (JButton) ad una finestra AWT rappresenta un'applicazione del pattern Decorator.

393. (2008-7-9)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Una classe astratta può avere campi istanza.
- LinkedList<String> è sottotipo di LinkedList<?>.
- String è assegnabile a Object.
- Un costruttore può chiamarne un altro della stessa classe usando la parola chiave this.
- Una classe interna può essere private.
- L'interfaccia Iterable contiene un metodo che restituisce un iteratore.
- Il pattern Composite prevede che un contenitore si possa comportare come un oggetto primitivo.
- Nel framework MVC, le classi model si occupano della presentazione dei dati agli utenti.
- Nel pattern Observer, l'osservatore ha un metodo per agganciarsi ad un oggetto da osservare.
- Il passaggio di un parametro Comparator al metodo Collections.sort rappresenta un’istanza del pattern Template Method.

394. (2008-6-19)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Si può effettuare l'overriding di un metodo statico.
- int è sottotipo di long.
- int è assegnabile a long.
- Una variabile locale può essere private.
- Si può scrivere “public class A<T, Integer> { }”.
- getClass è un metodo della classe Class.
- Il pattern Iterator prevede un metodo per rimuovere l'ultimo oggetto visitato.
- Nel framework MVC, le classi controller ricevono gli eventi dovuti all'interazione con l'utente.
- Nel pattern Observer, l'oggetto osservato ha un metodo per registrare un nuovo osservatore.
- La scelta del layout di un container AWT rappresenta un'istanza del pattern Strategy.

395. (2008-3-27)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un campo protected è visibile anche alle altre classi dello stesso pacchetto.
- La parola chiave synchronized si può applicare anche ad un campo.
- L'istruzione Number[] n = new Integer[10]; è corretta.
- L'istruzione LinkedList<Number> l = new LinkedList<Integer>(); è corretta.
- Si può effettuare l'overriding di un metodo statico.
- Un metodo statico può contenere una classe locale.
- Thread è un'interfaccia della libreria standard.
- Il pattern Composite prevede che sia gli oggetti primitivi sia quelli compositi implementino una stessa interfaccia.
- Nel pattern Observer, un oggetto può essere osservato da al più un osservatore.
- Nell'architettura Model-View-Controller, solo i controller dovrebbero modificare i modelli.

396. (2008-2-25)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Una classe final può estendere un'altra classe.
- Un parametro formale può essere dichiarato final.
- La parola chiave this può essere usata per chiamare un costruttore.
- La classe Thread è astratta.
- LinkedList<Integer> è sottotipo di LinkedList<Number>.
- LinkedList<Integer> è sottotipo di Iterable<Integer>.
- L'intestazione di classe class A<T,? extends T> è corretta.
- Un campo protected è visibile anche alle altre classi dello stesso pacchetto.

12 Vero o falso

- Il pattern **Template Method** si può applicare solo ad algoritmi che fanno uso di determinate operazioni primitive.
- Per aggiungere funzionalità a una classe A, il pattern **Decorator** suggerisce di creare una sottoclasse di A.
- Nel pattern **Composite**, sia gli oggetti primitivi che compositi implementano una stessa interfaccia.

397. (2008-1-30)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Un campo **protected** è visibile anche alle altre classi dello stesso pacchetto.
- Il costruttore **catch A** cattura le eccezioni di tipo A e delle sue sottoclassi.
- Un'interfaccia può avere un campo statico **final**.
- Un costruttore può essere dichiarato **final**.
- Applicato ad un metodo, il modificatore **final** impedisce sia l'overloading che l'overriding.
- Una classe non parametrica può implementare **Comparable<Integer>**.
- Un costruttore di una classe non parametrica può avere un parametro di tipo.
- La scelta del layout in un pannello Swing/AWT è un esempio del pattern **Strategy**.
- Per aggiungere funzionalità a una classe A, il pattern **Decorator** suggerisce di creare una sottoclasse di A.
- Nel pattern **Composite**, i client devono poter distinguere tra un oggetto primitivo e un oggetto composito.

398. (2007-9-17)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Se un metodo contiene una dichiarazione **throws**, ogni metodo che ne faccia l'overriding deve contenere una dichiarazione **throws**.
- L'eccezione **ArrayIndexOutOfBoundsException** non può essere catturata.
- Un metodo statico può accedere ai campi statici della classe.
- La classe **Class** è astratta.
- Una classe non parametrica può implementare **Collection<String>**.
- Nel binding dinamico, la lista delle firme candidate può essere vuota.
- Un metodo **synchronized** di un oggetto può essere chiamato da un solo thread per volta.
- Il pattern **Iterator** prevede un metodo per far avanzare di una posizione l'iteratore.
- Nel pattern **Observer**, gli osservatori devono contenere un riferimento all'oggetto osservato.
- Il pattern **Strategy** permette di fornire versioni diverse di un algoritmo.

399. (2007-7-20)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- OutOfMemoryError** è un'eccezione verificata.

- Il blocco `try { ... } catch (Exception e)` cattura anche `NullPointerException`.
- Una variabile locale può essere `private`.
- Se `T` è una variabile di tipo, si può scrivere `new LinkedList<T>()`.
- `HashSet<Integer>` è sottotipo di `Set<Integer>`.
- Un metodo statico può essere `abstract`.
- Un metodo non statico di una classe può chiamare un metodo statico della stessa classe.
- Nel pattern `Decorator`, l'oggetto decoratore si comporta come l'oggetto da decorare.
- Nel pattern `Decorator`, l'oggetto da decorare ha un metodo che aggiunge una decorazione.
- Il metodo `Collections.sort` rappresenta un'istanza del pattern `Strategy`.

400. (2007-6-29)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, l'esercizio vale 0.

Vero Falso

- Una classe può avere più metodi pubblici con lo stesso nome e lo stesso tipo restituito.
- Si può effettuare l'`overriding` di un costruttore.
- I costruttori possono sollevare eccezioni.
- Una classe `abstract` può avere campi.
- `LinkedList<Integer>` è sottotipo di `LinkedList<Number>`.
- Una classe anonima può avere costruttore.
- Un metodo pubblico di una classe può chiamare un metodo privato della stessa classe.
- Il pattern `Iterator` prevede che un iteratore abbia un metodo `remove`.
- Nell'architettura MVC, i controller non devono comunicare direttamente con i modelli.
- Nel pattern `Strategy`, si suggerisce di usare una classe per rappresentare un'algoritmo.

401. (2007-2-7)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- Una classe astratta può contenere campi.
- Una classe può essere privata (`private`).
- L'interfaccia `Iterator<Integer>` è sotto-interfaccia di `Iterator<Number>`.
- Si può dichiarare un riferimento di tipo `? (jolly)`.
- Si può dichiarare un riferimento di tipo `List<?>`.
- Una classe può avere un costruttore privato.
- L'istruzione “`String s;`” costruisce un oggetto di tipo `String`.
- La classe `Method` è sotto-classe di `Class`.
- Nella chiamata `a.f()`, le firme candidate sono ricercate a partire dalla classe dichiarata di `a`.
- `sleep` è un metodo statico della classe `Thread`.

402. (2007-2-23)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- Una variabile locale di un metodo può essere dichiarata **static**.
- Una classe astratta può implementare un'interfaccia.
- Un'interfaccia può avere un costruttore.
- Una classe deve necessariamente implementare un costruttore.
- Tutte le classi derivano da (cioè sono sottoclassi di) **Class**.
- Tutte le eccezioni derivano da **Exception**.
- clone** è un metodo pubblico di **Object**.
- Se **x** è un riferimento ad un **Employee**, l'istruzione “**x instanceof Object**” restituisce **false**.
- Se un thread sta eseguendo un metodo **synchronized** di un oggetto, nessun altro thread può eseguire i metodi di quell'oggetto.
- Una variabile locale di un metodo può essere dichiarata **private**.

403. (2007-1-12)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- Una classe può implementare più interfacce.
- Un'interfaccia può estendere una classe astratta.
- La classe **HashSet<Integer>** è sottoclasse di **HashSet<Number>**.
- Un metodo **static** può avere un parametro di tipo.
- L'interfaccia **List** estende **LinkedList**.
- L'interfaccia **Cloneable** contiene soltanto un metodo.
- La classe **Class** è astratta.
- La firma di un metodo comprende anche il tipo restituito dal metodo.
- Istanziare un oggetto della classe **Thread** provoca l'avvio immediato di un nuovo thread di esecuzione.
- Una classe può avere un costruttore privato.

404. (2006-9-15)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- L'interfaccia **Iterable** ha un parametro di tipo.
- La classe **String** ha un parametro di tipo.
- Il tipo jolly “**?**” è un parametro attuale di tipo.
- Una classe può avere più di un parametro di tipo.
- L'interfaccia **List** estende **Collection**.
- Le eccezioni derivate da **RuntimeException** non sono verificate.
- Gli oggetti di tipo **String** sono immutabili.
- Dato un insieme di firme di metodi, non sempre ce ne è una più specifica di tutte le altre.
- L'*early binding* è svolto dal programmatore.
- Il *late binding* è svolto dalla Java Virtual Machine.

405. (2006-7-17)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti, assenza di risposta 0 punti. Se il totale è negativo, vale 0.

Vero Falso

- Una interfaccia può avere metodi statici.
- Un campo final pubblico di una classe non viene ereditato.
- `LinkedList` implementa `Collection`.
- `Integer` è sottoclasse di `Number`.
- Java è un linguaggio funzionale.
- `OutOfMemoryError` è una eccezione non verificata.
- `Boolean` è sottoclasse di `Number`.
- `Boolean` è sottoclasse di `Object`.
- `sleep` è un metodo statico di `Runnable`.
- “`double d = 3;`” è una istruzione corretta.

406. (2006-6-26)

Dire quali delle seguenti affermazioni sono vere, e quali false. Valutazione: risposta giusta +2 punti, risposta errata -2 punti. Se il totale è negativo, vale 0.

Vero Falso

- Una interfaccia può estendere una classe.
- Una interfaccia può estendere un’altra interfaccia.
- Una interfaccia può avere campi.
- `Integer` è sottoclasse di `Number`.
- Linguaggi I è propedeutico per Linguaggi II .
- `RuntimeException` è una eccezione non verificata.
- `String` è sottoclasse di `Object`.
- Ogni oggetto `Thread` corrisponde ad un thread di esecuzione.
- I metodi `equals` e `hashCode` di `Object` sono coerenti tra loro.
- “`Double d = 3;`” è una istruzione corretta.

13 Lambda-espressioni

407. (**Lambda, 2019-4-29**)

Data la seguente interfaccia funzionale:

```
public interface Predicate<T> {  
    boolean test(T t);  
}
```

usare una lambda-espressione per definire ciascuna delle seguenti implementazioni dell'interfaccia:

- a) Un predicato che accetta un intero e restituisce true se è pari.
- b) Un predicato che accetta una stringa e restituisce true se quella stringa è uguale a "exit".
- c) Un predicato che accetta un SortedSet<Integer> e restituisce true se l'elemento minimo dell'insieme è negativo e quello massimo è positivo.
- d) Un predicato che accetta una Collection<Object> e restituisce true se la collezione contiene almeno un duplicato (secondo equals).

408. (**2019-10-9**)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A {  
    Comparator<Double> b = new B(null);  
  
    Comparator<String> c = (x, y) -> B.g(x, y);  
  
    <T> A f(T x, T y) {  
        return new B(x==y);  
    }  
}
```


14 Clonazione di oggetti

409. (Book, 2016-7-21)

Implementare la classe Book, che rappresenta un libro diviso in capitoli. Il metodo addChapter aggiunge un capitolo in coda al libro, caratterizzato da titolo e contenuto. I capitoli sono automaticamente numerati a partire da 1. Il metodo getChapterName(i) restituisce il titolo del capitolo *i*-esimo, mentre il metodo getChapterContent(i) ne restituisce il contenuto.

Gli oggetti Book devono essere clonabili. Inoltre, la classe deve essere dotata di ordinamento naturale, basato sul numero di capitoli.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|---|
| Esempio d'uso: | Output: |
| <pre>Book b = new Book(); b.addChapter("Prefazione", "Sono_passati_pochi_anni..."); b.addChapter("Introduzione", "Un_calcolatore_digitale...") ; b.addChapter("Sistemi_di_elaborazione", "Un_calcolatore... "); Book bb = b.clone(); System.out.println(bb.getChapterContent(1)); System.out.println(bb.getChapterTitle(2));</pre> | Sono passati pochi anni... Introduzione |

410. (Insieme di lettere, 2013-1-22)

La classe MyString rappresenta una stringa. Due oggetti di tipo MyString sono considerati uguali (da `equals`) se utilizzano le stesse lettere, anche se in numero diverso. Ad esempio, “casa” è uguale a “cassa” e diverso da “sa”; “culle” è uguale a “luce” e diverso da “alluce”. La classe MyString deve essere clonabile e deve offrire un’implementazione di `hashCode` coerente con `equals` e non banale (che non restituisca lo stesso codice hash per tutti gli oggetti).

Suggerimento: Nella classe String è presente il metodo `public char charAt(int i)`, che restituisce l’*i*-esimo carattere della stringa, per i compreso tra 0 e `length()`-1.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>MyString a = new MyString("freddo"); MyString b = new MyString("defro"); MyString c = new MyString("caldo"); MyString d = c.clone(); System.out.println(a.equals(b)); System.out.println(b.equals(c)); System.out.println(a.hashCode() == b.hashCode());</pre> | true false true |

411. (Anagrammi, 2012-9-3)

Implementare la classe MyString, che rappresenta una stringa con la seguente caratteristica: due oggetti MyString sono considerati uguali (da `equals`) se sono uno l'anagramma dell'altro. Inoltre, la classe MyString deve essere clonabile e deve offrire un’implementazione di `hashCode` coerente con `equals` e non banale (che non restituisca lo stesso codice hash per tutti gli oggetti).

Suggerimento: Nella classe String è presente il metodo `public char charAt(int i)`, che restituisce l’*i*-esimo carattere della stringa, per i compreso tra 0 e `length()`-1.

| | |
|---|---|
| Esempio d'uso: | Output dell'esempio d'uso: true false true |
| <pre>MyString a = new MyString("uno_due_tre"); MyString b = new MyString("uno_tre_deu"); MyString c = new MyString("ert_unodue"); MyString d = c.clone(); System.out.println(a.equals(b)); System.out.println(b.equals(c)); System.out.println(a.hashCode()==b.hashCode());</pre> | |

412. (Segment, 2010-11-30)

Implementare la classe **Segment**, che rappresenta un segmento collocato nel piano cartesiano. Il costruttore accetta le coordinate dei due vertici, nell'ordine x_1, y_1, x_2, y_2 . Il metodo **getDistance** restituisce la distanza tra la retta che contiene il segmento e l'origine del piano. Ridefinire il metodo **equals** in modo che due segmenti siano considerati uguali se hanno gli stessi vertici. Fare in modo che i segmenti siano clonabili.

Si ricordi che:

- L'area del triangolo con vertici di coordinate (x_1, y_1) , (x_2, y_2) e (x_3, y_3) è data da:

$$\frac{|x_1(y_2 - y_3) - x_2(y_1 - y_3) + x_3(y_1 - y_2)|}{2}.$$

| | |
|--|---|
| Esempio d'uso: | Output dell'esempio d'uso: true 2.4 |
| <pre>Segment s1 = new Segment(0.0, -3.0, 4.0, 0.0); Segment s2 = new Segment(4.0, 0.0, 0.0, -3.0); Segment s3 = s2.clone(); System.out.println(s1.equals(s2)); System.out.println(s1.getDistance());</pre> | |

413. (2010-1-22)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente e permetta la compilazione della classe A.

```
public class A extends B {

    public A() {
        b1 = new B.C(true);
        b2 = new B(false);
    }

    public B f(Object o) {
        B x = super.f(o);
        return x.clone();
    }

    private B.C c = new B.C(3);
    private B b1, b2;
}
```

414. (TreeType, 2006-9-15)

Implementare le classi **TreeType** e **Tree**. **TreeType** rappresenta un tipo di albero (pino, melo, etc.), mentre **Tree** rappresenta un particolare esemplare di albero. Ogni **TreeType** è caratterizzato dal suo nome. Ogni **Tree** ha un tipo base ed eventualmente degli innesti di altri tipi di alberi. Il metodo **addGraft** di **Tree** aggiunge un innesto ad un albero, purchèE non sia dello stesso tipo dell'albero stesso. Il metodo **getCounter** di **Tree** restituisce il numero di alberi che sono stati creati. Il metodo **getCounter** di **TreeType** restituisce il numero di alberi di quel tipo che sono stati creati. (32 punti)

Ridefinire il metodo `clone` di `Tree`, facendo attenzione ad eseguire una copia profonda laddove sia necessario. (8 punti)

Esempio d'uso:

```
TreeType melo = new TreeType("melo"
);
TreeType pero = new TreeType("pero")
;
Tree unMelo = new Tree(melo);
Tree unAltroMelo = new Tree(melo);

unAltroMelo.addGraft(pero);
unAltroMelo.addGraft(pero);
System.out.println("Sono stati creati "
+ melo.getCounter() + " meli fino "
+a questo momento.");
System.out.println("Sono stati creati "
+ Tree.getCounter() + " alberi fino "
+a questo momento.");
System.out.println(unAltroMelo);
unAltroMelo.addGraft(melo);
```

Output dell'esempio d'uso:

```
Sono stati creati 2 meli fino a questo momento.
Sono stati creati 2 alberi fino a questo momento.
```

```
tipo: melo
innesti:
pero
```

```
Exception in thread "main":
java.lang.RuntimeException
```


15 Riflessione

415. (2022-1-26)

Determinare l'output del seguente frammento di codice Java:

```
Short s = 10;
Integer a = 10;
Float f = new Float(10f);
Number aa = a;
Object o = f;

System.out.println((Object)s == (Object)a);
System.out.println((s+1) == (a+1));
System.out.println(aa == a);
System.out.println(a.getClass().equals(aa.getClass()));
System.out.println(o.getClass().equals(f.getClass()));
System.out.println(o.getClass().equals(Object.class));
```

416. (2021-10-26)

Determinare l'output del seguente frammento di codice Java:

```
Short s = 10;
Integer a = 10;
Double b = 11.0;
Number aa = a;
Number bb = b;

System.out.println( ((Object) s) == ((Object) a) );
System.out.println(aa == a);
System.out.println(a.getClass().equals(aa.getClass()));
System.out.println(aa.getClass().equals(bb.getClass()));
Object o = s;
System.out.println(o.getClass().equals(Short.class));
```

417. (GetType, 2010-1-22)

Implementare il metodo statico `GetType` che, data una collezione `c` (`Collection`) ed un oggetto `x` di tipo `Class`, restituisce un oggetto della collezione il cui tipo effettivo sia esattamente `x`. Se un tale oggetto non esiste, il metodo restituisce `null`.

Prestare particolare attenzione alla scelta della firma del metodo. Si ricordi che la classe `Class` è parametrica.

418. (CountGetType, 2009-11-27)

Implementare il metodo statico `countGetType` che, data una lista di oggetti, stampa a video il numero di oggetti contenuti nella lista, *divisi in base al loro tipo effettivo*.

Attenzione: il metodo deve funzionare con qualunque tipo di lista e di oggetti contenuti.

Esempio d'uso:

```
List<Number> l = new LinkedList<Number>();
l.add(new Integer(3));
l.add(new Double(4.0))
l.add(new Float(7.0f));
l.add(new Integer(11));
countGetType(l);
```

Output dell'esempio d'uso:
java.lang.Double : 1
java.lang.Float : 1
java.lang.Integer : 2

419. (SuperclassIterator, 2006-9-15)

Implementare una classe SuperclassIterator che rappresenta un iteratore su tutte le superclassi di un oggetto dato, a partire dalla classe stessa dell'oggetto fino ad arrivare ad Object.

Ad esempio, nell'ambito della tradizionale gerarchia formata dalle classi Employee e Manager, si consideri il seguente caso d'uso.

| | |
|---|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Iterator<Class<?>> i = new SuperclassIterator(new Manager("Franco")); while (i.hasNext()) System.out.println(i.next());</pre> | <pre>class Manager class Employee class java.lang.Object</pre> |

16 Multi-threading

420. (parallelMax, 2022-3-28)

Implementare il metodo statico `parallelMax`, che accetta due iteratori e un comparatore, e restituisce l'elemento massimo, secondo il comparatore, tra tutti quelli restituiti dai due iteratori. Il metodo deve usare due thread in parallelo: ciascuno scorre uno dei due iteratori.

Prestare particolare attenzione alla scelta della firma del metodo.

421. (MysteryThread9, 2021-9-24)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) throws InterruptedException {
    final Object x = new Object();
    final int[] count = new int[1];

    class MyThread extends Thread {
        int id;
        MyThread(int n) { id = n; }
        public void run() {
            synchronized (x) {
                count[0]++;
                synchronized (count) {
                    count.notify();
                }
                try {
                    x.wait();
                } catch (Exception e) { }
            }
            System.out.println(id);
        }
    }
    Thread t1 = new MyThread(1), t2 = new MyThread(2), t3 = new MyThread(3);
    t1.start(); t2.start(); t3.start();
    synchronized (count) {
        while (count[0]<3) {
            count.wait();
            System.out.println("Incremento");
        }
    }
    System.out.println("Fatto");
    synchronized (x) {
        x.notifyAll();
    }
}
```

422. (2021-7-26)

Il seguente thread accede ad una lista di Employee, precedentemente istanziata. Gli oggetti Employee hanno un campo salario (*salary*) e un campo anzianità in servizio (*years*). Un thread di tipo AgeBonus assegna un bonus di 100 euro agli impiegati la cui anzianità supera una data soglia.

```
public class AgeBonus extends Thread {
    private final int threshold;
    public AgeBonus(int n) { this.threshold = n; }
```

```

@Override
public void run() {
    for (Employee e: list) {
        if (e.getYears() > threshold)
            e.setSalary(e.getSalary() + 100);
    }
}

```

Dire quali dei seguenti inserimenti consentono a un programma di eseguire concorrentemente un numero arbitrario di thread `AgeBonus`, senza incorrere in *race condition* (è possibile indicare più risposte, intese come alternative).

In aggiunta, dire quale inserimento è quello migliore e perché.

- | | | |
|-----|-------------------------------------|---|
| (a) | 1 = “ synchronized (this){ ” | 4 = “ } ” |
| (b) | 1 = “ synchronized { ” | 4 = “ } ” |
| (c) | 1 = “ synchronized (list){ ” | 4 = “ } ” |
| (d) | 2 = “ synchronized (this){ ” | 3 = “ } ” |
| (e) | 2 = “ synchronized (list){ ” | 3 = “ } ” |
| (f) | 2 = “ synchronized (e){ ” | 3 = “ } ” |
| (g) | 2 = “ list .wait(); ” | 3 = “ list .notifyAll(); ” |
| (h) | 1 = “ synchronized(list){ ” | 2 = “ list .wait(); ” 3 = “ list .notifyAll(); ” 4 = “ } ” |

423. (Missing synch 4, 2021-10-26)

Le istanze della seguente classe `MyThread` condividono due array di interi (`int[]`), `a` e `b`, precedentemente istanziati e inizializzati.

```

class MyThread extends Thread {
    public void run() {
        for (int i=0; i<a.length; i++) {
            if (a[i] > b[i]) {
                int temp = b[i];
                b[i] = a[i];
                a[i] = temp;
            }
        }
    }
}

```

Un programma avvia *due* thread di tipo `MyThread`, con l'obiettivo che, dopo l'esecuzione, ciascun elemento di `a` sia minore o uguale del corrispondente elemento di `b`.

Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- | | | |
|-----|---|-----------------------------------|
| (a) | non è necessario aggiungere nulla | |
| (b) | <code>1 = "synchronized (this){"</code> | <code>4 = "}"</code> |
| (c) | <code>1 = "synchronized (MyThread.class){"</code> | <code>4 = "}"</code> |
| (d) | <code>1 = "synchronized {"</code> | <code>4 = "}"</code> |
| (e) | <code>1 = "synchronized (a){"</code> | <code>4 = "}"</code> |
| (f) | <code>1 = "synchronized (b){"</code> | <code>4 = "}"</code> |
| (g) | <code>2 = "synchronized (this){"</code> | <code>3 = "}"</code> |
| (h) | <code>2 = "synchronized (a[i]){"</code> | <code>3 = "}"</code> |
| (i) | <code>2 = "synchronized (b){"</code> | <code>3 = "}"</code> |
| (j) | <code>2 = "a.wait();"</code> | <code>3 = "a.notifyAll();"</code> |

424. (MysteryThread8, 2020-2-27)

Elencare tutte le sequenze di output possibili per il seguente programma.

```
public class A {
    private volatile int n;
    public int incrementAndGet() {
        return ++n;
    }

    public static void main(String[] args) {
        A a = new A(), b = new A();
        Thread t1 = new Thread(() -> System.out.println(a.incrementAndGet())),
            t2 = new Thread(() -> System.out.println(b.incrementAndGet())),
            t3 = new Thread(() -> System.out.println(a.incrementAndGet()));
        t1.start(); t2.start(); t3.start();
    }
}
```

425. (MysteryThread7, 2020-1-24)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) throws InterruptedException {
    final Object x = new Object();
    final int[] count = new int[1]; // Don't do this: use AtomicInteger

    class MyThread extends Thread {
        public void run() {
            synchronized (x) {
                count[0]++;
                x.notify();
                System.out.println(count[0]);
            }
        }
    }
    Thread t1 = new MyThread(), t2 = new MyThread(), t3 = new MyThread();
    t1.start(); t2.start(); t3.start();
    synchronized (x) {
        count[0] = -1;
        while (count[0]<0) x.wait();
    }
    t2.join();
    System.out.println("Fatto");
}
```

426. (MysteryThread6, 2019-9-20)

(a) Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) throws InterruptedException {
    final Object x = new Object(), y = new Object();
    Thread t1 = new Thread(() -> {
        synchronized (x) {
            try {
                x.wait();
                synchronized (y) {
                    y.notify();
                }
            }
            catch (Exception e) { return; }
            finally { System.out.println("t1"); }
        }
    });
    Thread t2 = new Thread(() -> {
        synchronized (y) {
            try { y.wait(); }
            catch (Exception e) { return; }
            finally { System.out.println("t2"); }
        }
    });
    t1.start();
    t2.start();
    synchronized (y) {
        y.notify();
    }
    System.out.println("main");
}

```

(b) Come cambiano le sequenze di output se le ultime righe del `main` vengono modificate come segue?

```

synchronized (y) {
    y.notify();
    System.out.println("main");
}

```

427. (RandomExecutor, 2019-7-23)

Implementare la classe `RandomExecutor` che esegue degli oggetti `Runnable` sequenzialmente e in ordine casuale. Precisamente, la classe offre un costruttore senza argomenti e i seguenti metodi:

- Il metodo `void addTask(Runnable r)` aggiunge un task. Questo metodo può essere chiamato solo prima di `launch`.
- Il metodo `void launch()` avvia l'esecuzione dei task. Questo metodo non è bloccante.
- Il metodo bloccante `void join(Runnable r)` attende la terminazione del corrispondente task. Questo metodo può essere chiamato prima o dopo `launch`.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|---|---|
| Esempio d'uso: | Output: I numeri da 1 a 3 in ordine casuale. |
| <pre> RandomExecutor e = new RandomExecutor(); Runnable r1 = () -> { System.out.println(1); }; Runnable r2 = () -> { System.out.println(2); }; Runnable r3 = () -> { System.out.println(3); }; e.addTask(r1); e.addTask(r2); e.addTask(r3); e.launch(); e.join(r2); </pre> | |

428. (Missing synch 3, 2019-3-19)

Le istanze di MyThread condividono un array di interi **a** (tipo **int[]**), precedentemente istanziato e inizializzato.

```
class MyThread extends Thread {  
    public void run() {  
        final int n = a.length - 1;  
        for (int i=0; i <= n/2; i++) {  
            // scambia a[i] e a[n-i]  
            int temp = a[i];  
            a[i] = a[n - i];  
            a[n - i] = temp;  
        }  
    }  
}
```

Se un programma avvia *un solo* thread di tipo MyThread, questo rovescerà il contenuto dell'array **a**.

Un programma avvia *due* thread di tipo MyThread, con l'intento di ritrovare l'array **a** inalterato. Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- | | |
|--|----------------------------|
| (a) non c'è bisogno di aggiungere nulla | |
| (b) aggiungere il modificatore synchronized al metodo run | |
| (c) 1 = “ synchronized (this){ ” | 4 = “ } ” |
| (d) 1 = “ synchronized { ” | 4 = “ } ” |
| (e) 1 = “ synchronized (a){ ” | 4 = “ } ” |
| (f) 1 = “ synchronized (MyThread.class){ ” | 4 = “ } ” |
| (g) 1 = “ a.wait(); ” | 4 = “ a.notify(); ” |
| (h) 2 = “ synchronized (this){ ” | 3 = “ } ” |
| (i) 2 = “ synchronized (a[i]){ ” | 3 = “ } ” |
| (j) 2 = “ synchronized (a){ ” | 3 = “ } ” |
| (k) 2 = “ synchronized (MyThread.class){ ” | 3 = “ } ” |

429. (Missing synch 2, 2019-2-15)

Le istanze di MyThread condividono due array di interi, **a** e **b**, precedentemente istanziati e inizializzati.

```
class MyThread extends Thread {  
    public void run() {  
        for (int i=0; i < a.length; i++) {  
            if (a[i]>0) {  
                b[i] = a[i];  
                a[i] = 0;  
            }  
        }  
    }  
}
```

Un programma avvia due thread di tipo MyThread, con l'obiettivo di spostare i valori positivi da **a** a **b**.

Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- | | | |
|-----|---|--------------------------------|
| (a) | non c'è bisogno di aggiungere nulla | |
| (b) | <code>1 = "synchronized (this){"</code> | <code>4 = "}"</code> |
| (c) | <code>1 = "synchronized {"</code> | <code>4 = "}"</code> |
| (d) | <code>1 = "synchronized (a){"</code> | <code>4 = "}"</code> |
| (e) | <code>2 = "synchronized (this){"</code> | <code>3 = "}"</code> |
| (f) | <code>2 = "synchronized (a[i]){"</code> | <code>3 = "}"</code> |
| (g) | <code>2 = "synchronized (b){"</code> | <code>3 = "}"</code> |
| (h) | <code>2 = "a.wait();"</code> | <code>3 = "a.notify();"</code> |

430. (Shop, 2019-10-9)

Implementare la classe thread-safe `Shop<T>` che rappresenta un negozio di oggetti di tipo `T`, e fornisce i seguenti metodi:

- **public void sell(T object, int price)**
Mette in vendita un oggetto ad un dato prezzo
- **public T buy(int offer)**
Mette in attesa il chiamante finché non ci sia un oggetto in vendita il cui prezzo è inferiore o uguale all'offerta; a quel punto rimuove quell'oggetto dalla vendita e lo restituisce al chiamante

Nota: se viene invocato due volte `sell` con lo stesso oggetto, senza che sia stato venduto nel frattempo, il secondo prezzo deve sostituire il primo.

431. (GuessTheNumber, 2019-1-23)

Realizzare la classe `GuessTheNumber`, che consente a diversi thread di indovinare un numero segreto. Il costruttore accetta il numero da indovinare (`int`) e la durata del quiz, in millisecondi (`long`). Il metodo `guessAndWait` consente ad un thread di proporre una soluzione (`int`), poi attende fino alla fine del quiz, ed infine restituisce `true` se questo è il thread che si è avvicinato di più alla soluzione, e `false` altrimenti.

Rispondere alle seguenti domande relative alla vostra implementazione:

- Cosa succede se più thread arrivano a pari merito?
- Cosa succede se un thread chiama `guessAndWait` quando il quiz è già terminato?

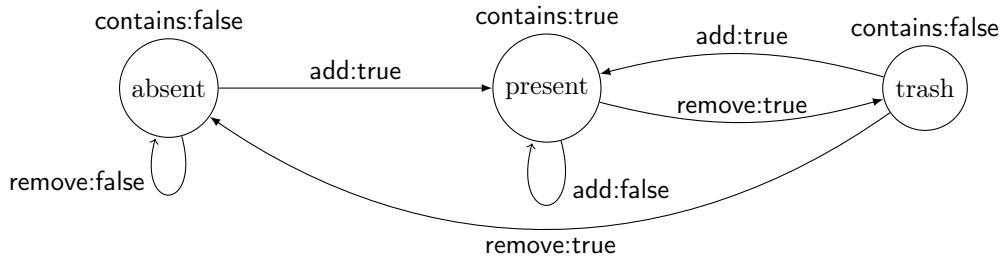
432. (SharedCounter, 2018-9-17)

Realizzare la classe `SharedCounter`, che rappresenta un contatore *thread-safe*, che parte dal valore zero. Il suo metodo `incr` incrementa il contatore, mentre `decr` lo decrementa. Il contatore non può assumere un valore negativo, quindi invocare `decr` quando il valore è zero non ha alcun effetto.

Il metodo `waitForValue` accetta un valore intero `n` e mette il chiamante in attesa finché il contatore non assuma il valore `n` (se il contatore vale già `n`, il metodo restituisce subito il controllo al chiamante).

433. (SafeSet, 2018-7-19)

Realizzare la classe `SafeSet`, che rappresenta un insieme che richiede due passaggi per rimuovere completamente un oggetto. Il metodo `add` aggiunge un elemento all'insieme, restituendo `true` se l'inserimento ha avuto successo. Il metodo `remove` rimuove un elemento dall'insieme, ma la rimozione è definitiva solo dopo una seconda chiamata. Il metodo `contains` verifica se l'insieme contiene un dato elemento (in base a `equals`). Infine, un `SafeSet` deve essere *thread-safe*. Il seguente diagramma rappresenta il ciclo di vita di un oggetto all'interno di un `SafeSet`:



L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```

SafeSet<String> a = new SafeSet<>();
System.out.println(a.add("ciao"));
System.out.println(a.add("mondo"));
System.out.println(a.remove("ciao"));
System.out.println(a.contains("ciao"));
System.out.println(a.remove("ciao"));
System.out.println(a.contains("ciao"));

```

Output:

```

true
true
true
false
true
false

```

434. (MysteryThread5, 2018-7-19)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) throws InterruptedException {
    final Object x = new Object();
    final int[] count = new int[1]; // don't do this: use AtomicInteger

    class MyThread extends Thread {
        int id;
        MyThread(int n) { id = n; }
        public void run() {
            synchronized (x) {
                synchronized (count) {
                    count[0]++;
                    count.notify();
                }
                try {
                    x.wait();
                } catch (Exception e) {
                    x.notify();
                } finally {
                    System.out.println(id);
                }
            }
        }
    }
    Thread t1 = new MyThread(1), t2 = new MyThread(2), t3 = new MyThread(3);
    t1.start(); t2.start(); t3.start();
    synchronized (count) {
        while (count[0]<3) count.wait();
    }
    t2.interrupt();
    t2.join();
    System.out.println("Fatto");
}

```

435. (PeriodicExecutor, 2018-6-20)

Realizzare la classe PeriodicExecutor, che consente di eseguire una serie di task periodicamente, con un limite al numero di task che possono essere eseguiti contemporaneamente. Il costruttore

accetta questo limite. Il metodo `addTask` accetta un `Runnable` e un `long x`, e fa in modo che il `Runnable` venga eseguito ripetutamente, con un ritardo di `x` millisecondi tra la fine di un'esecuzione e l'inizio della successiva.

Se però (ri)avviare un `Runnable` porta a superare il limite, l'avvio viene rimandato finché ci sarà la possibilità di eseguirlo senza violare il limite.

Il limite si riferisce al numero di task che stanno eseguendo il loro `Runnable`, non al periodo durante il quale stanno aspettando il ritardo `x`.

L'implementazione deve rispettare il seguente esempio d'uso.

```
PeriodicExecutor exec = new PeriodicExecutor(2);
Runnable r1 = ..., r2 = ..., r3 = ...;
exec.addTask(r1, 1000);
exec.addTask(r2, 500);
exec.addTask(r3, 700);
```

436. (Two threads, 2018-2-22)

Implementare un programma Java che avvia contemporaneamente due thread, che condividono una lista di interi:

- il primo thread aggiunge un numero casuale alla lista ogni decimo di secondo, terminando quando il numero estratto è multiplo di 100;
- il secondo thread calcola e stampa a video la somma di tutti i numeri nella lista, una volta al secondo, terminando non appena termina il primo thread.

È necessario evitare *race condition* e attese attive.

Suggerimento. Per ottenere un numero intero casuale, si consiglia di utilizzare le seguenti funzionalità della classe `java.util.Random`:

```
public Random()
public int nextInt()
```

437. (Shared total, 2018-1-24)

Implementare la classe `SharedTotal`, che permette a diversi thread di comunicare un valore numerico (`double`) e poi ricevere la somma di tutti i valori inviati dai diversi thread. Precisamente, il costruttore accetta come argomento un *timeout* in millisecondi. Il metodo `sendValAndReceiveTot` accetta come argomento il valore indicato dal thread corrente, mette il thread corrente in attesa fino allo scadere del timeout, e infine restituisce il totale di tutti i valori inviati.

Se un thread ha già chiamato `sendValAndReceiveTot` una volta, al secondo tentativo viene sollevata un'eccezione.

È necessario evitare *race condition*.

Esempio d'uso:

```
SharedTotal tot = new SharedTotal(1000);
try {
    System.out.println(tot.sendValAndReceiveTot(5.0));
} catch (InterruptedException e) {
    ...
}
```

Comportamento:

In assenza di altri thread, dopo un secondo stamperà 5.0

438. (MysteryThread4, 2017-7-20)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) {
    class MyThread extends Thread {
        private int id;
        private Object object;
```

```

public MyThread(int n, Object x) {
    id = n;
    object = x;
}
public void run() {
    try {
        synchronized (object) {
            object.wait();
        }
    } catch (InterruptedException e) {
        return;
    }
    System.out.println(id);
}
Object o1 = new Object(), o2 = new Object();
Thread t1 = new MyThread(1,o1);
Thread t2 = new MyThread(2,o1);
Thread t3 = new MyThread(3,o1);
Thread t4 = new MyThread(4,o2);
t1.start(); t2.start(); t3.start(); t4.start();
try { Thread.sleep(1000); } catch (InterruptedException e) { }
synchronized (o2) { o2.notifyAll(); }
synchronized (o1) { o1.notify(); }
}

```

439. (Market, 2017-6-21)

Realizzare la classe *thread-safe* Market, che permette a diversi thread di contrattare il prezzo di un servizio. Il metodo `sell` offre il servizio per un certo prezzo e blocca il chiamante finché non arriverà una richiesta con un importo adeguato. Simmetricamente, il metodo `buy` richiede il servizio per un certo prezzo e blocca il chiamante finché non arriverà un'offerta adeguata.

Per semplicità, si può supporre che tutti gli importi passati a `sell` e `buy` siano diversi.

L'implementazione deve rispettare il seguente esempio d'uso, in cui diversi thread condividono un oggetto Market `m`:

| | | |
|-----------|-----------------------------|--------------------------------|
| Thread 1: | <code>m.buy(10.0);</code> | resta bloccato |
| Thread 2: | <code>m.sell(15.50);</code> | resta bloccato |
| Thread 3: | <code>m.sell(12.0);</code> | resta bloccato |
| Thread 4: | <code>m.buy(13.0);</code> | sblocca T3 e ritorna |
| Thread 5: | <code>m.buy(11.0);</code> | resta bloccato |
| Thread 6: | <code>m.sell(9.50);</code> | sblocca T1 oppure T5 e ritorna |

440. (Bonus per Employee, 2017-3-23)

I seguenti thread accedono ad una lista di Employee, precedentemente istanziata. Gli oggetti Employee hanno un campo salario (*salary*) e un campo anzianità in servizio (*years*). Il primo thread assegna un bonus di 150 agli impiegati in servizio da più di 10 anni. Il secondo thread assegna un bonus di 200 agli impiegati che hanno un salario inferiore a 1500.

| | |
|--|---|
| <pre> class AgeBonus extends Thread { public void run() { for (Employee e: list) { if (e.getYears()>10) e.setSalary(e.getSalary()+150); } } } </pre> | <pre> class LowBonus extends Thread { public void run() { for (Employee e: list) { if (e.getSalary()<1500) e.setSalary(e.getSalary()+200); } } } </pre> |
|--|---|

Se un programma avvia questi due thread, dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte, intese come alternative).

Inoltre, se si sceglie più di una risposta, commentare sinteticamente la performance che si otterrebbe con ciascuna di esse.

- (a) 1 = “**synchronized (this){**” 4 = “**}**”
- (b) 1 = “**synchronized {**” 4 = “**}**”
- (c) 1 = “**synchronized (list){**” 4 = “**}**”
- (d) 2 = “**synchronized (this){**” 3 = “**}**”
- (e) 2 = “**synchronized (list){**” 3 = “**}**”
- (f) 2 = “**synchronized (e){**” 3 = “**}**”
- (g) 2 = “list .wait();” 3 = “list .notify();”
- (h) 1 = “**synchronized {**” 2 = “list .wait();” 3 = “list .notify();” 4 = “**}**”

441. (sumAndMax, 2017-2-23)

Implementare il metodo statico **sumAndMax**, che accetta un array di **double** e restituisce un array di due **double**, che conterranno rispettivamente la somma (anche parziale) andasse in overflow, il metodo deve interrompere il thread che sta calcolando il massimo e restituire **null**.

Suggerimento: se un’addizione tra due **double** va in overflow, il suo risultato sarà **Double.POSITIVE_INFINITY** o **Double.NEGATIVE_INFINITY**.

442. (Somma e azzera, 2017-10-6)

Il seguente thread accede ad un array di interi, precedentemente istanziato.

```

class MyThread extends Thread {
    public void run() {
        int tot = 0;
        for (int i=0; i<array.length; i++) {
            tot += array[i];
            array[i] = 0;
        }
        System.out.println(tot);
    }
}

```

Un programma avvia due thread di tipo **MyThread**, con l’obiettivo di ottenere l’output

```

<totale dell’array>
0

```

Dire quali dei seguenti inserimenti garantiscono l'output desiderato (è possibile indicare più risposte):

- (a) 1 = “**synchronized** (**this**){}” 4 = “}”
- (b) 1 = “**synchronized** {}” 4 = “}”
- (c) 1 = “**synchronized** (**array**){}” 4 = “}”
- (d) 1 = “**synchronized** (**tot**){}” 4 = “}”
- (e) 2 = “**synchronized** (**this**){}” 3 = “}”
- (f) 2 = “**synchronized** (**array**){}” 3 = “}”

443. (**mergeIfSorted**, 2017-1-25)

Implementare il metodo statico **mergeIfSorted**, che accetta due liste *a* e *b*, e un comparatore *c*, e restituisce un'altra lista. Inizialmente, usando due thread diversi, il metodo verifica che le liste *a* e *b* siano ordinate in senso non decrescente (ogni thread si occupa di una lista). Poi, se le liste sono effettivamente ordinate, il metodo le fonde (senza modificarle) in un'unica lista ordinata, che viene restituita al chiamante. Se, invece, almeno una delle due liste non è ordinata, il metodo termina restituendo **null**.

Il metodo dovrebbe avere complessità di tempo lineare.

Porre particolare attenzione alla scelta della firma, considerando i criteri di funzionalità, completezza, correttezza, fornitura di garanzie e semplicità.

444. (**Somma due**, 2016-9-20)

Il seguente thread accede ad un array di interi, precedentemente istanziato.

```
class MyThread extends Thread {  
    public void run() {  
        -----  
        for (int i=0; i<array.length; i++) {  
            -----  
            array[i]++;  
            -----  
        }  
        -----  
    }  
}
```

Un programma avvia due thread di tipo **MyThread**, con l'obiettivo di incrementare ogni elemento dell'array di 2. Dire quali dei seguenti inserimenti rendono il programma corretto ed esente da *race condition* (è possibile indicare più risposte):

- (a) 1 = “**synchronized** (**this**){}” 4 = “}”
- (b) 1 = “**synchronized** {}” 4 = “}”
- (c) 1 = “**synchronized** (**array**){}” 4 = “}”
- (d) 2 = “**synchronized** (**this**){}” 3 = “}”
- (e) 2 = “**synchronized** (**array**){}” 3 = “}”
- (f) 2 = “**array.wait()**;” 3 = “**array.notify()**;”

445. (**findString**, 2016-7-21)

Implementare il metodo statico **findString** che accetta una stringa *x* e un array di stringhe *a* e restituisce “vero” se *x* è una delle stringhe di *a*, e “falso” altrimenti. Per ottenere questo risultato, il metodo usa due tecniche in parallelo: un primo thread confronta *x* con ciascuna stringa dell'array; un altro thread confronta solo la lunghezza di *x* con quella di ciascuna stringa dell'array. Il metodo deve restituire il controllo al chiamante appena è in grado di fornire una risposta certa.

Ad esempio, se il secondo thread scopre che nessuna stringa dell'array ha la stessa lunghezza di *x*, il metodo deve subito restituire “falso” e terminare il primo thread (se ancora in esecuzione).

446. (**BlockingArray**, 2016-6-22)

Realizzare la classe **BlockingArray**, che rappresenta un array di dimensione fissa, in cui diversi thread mettono e tolgono elementi.

Il costruttore accetta la dimensione dell'array. Inizialmente, tutte le celle risultano vuote. Il metodo `put(i, x)` inserisce l'oggetto `x` nella cella `i`-esima, aspettando se quella cella è occupata. Simmetricamente, il metodo `take(i)` preleva l'oggetto dalla cella `i`-esima, aspettando se quella cella è vuota. La classe deve risultare *thread-safe*.

L'implementazione deve rispettare il seguente esempio d'uso.

| | |
|--|--|
| Esempio d'uso: | Output: <code>uno</code> A questo punto il thread si ferma finché un altro thread non invocherà <code>take(1)</code> |
| <pre>BlockingArray<String> array = new BlockingArray <>(10); array.put(0, "uno"); array.put(1, "due"); System.out.println(array.take(0)); array.put(1, "tre");</pre> | |

447. (MysteryThread3, 2016-3-3)

Escludendo i cosiddetti *spurious wakeup*, elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) {
    class MyThread extends Thread {
        private int id;
        private Object object;
        public MyThread(int n, Object x) {
            id = n;
            object = x;
        }
        public void run() {
            try {
                if (object!=null) synchronized (object) {
                    object.wait();
                }
            } catch (InterruptedException e) { return; }
            System.out.println(id);
        }
    }
    Object o1 = new Object(), o2 = new Object();
    Thread t1 = new MyThread(1,null);
    Thread t2 = new MyThread(2,o1);
    Thread t3 = new MyThread(3,o1);
    Thread t4 = new MyThread(4,o2);
    t1.start(); t2.start(); t3.start(); t4.start();
    try { Thread.sleep(1000); } catch (InterruptedException e) { }
    synchronized (o2) { o2.notify(); }
    synchronized (o1) { o1.notify(); }
}
```

448. (twoPhases, 2016-1-27)

Implementare il metodo statico `twoPhases`, che accetta due `Iterable<Runnable>` ed esegue in parallelo tutti i `Runnable` contenuti nel primo `Iterable` (prima fase), seguiti da tutti i `Runnable` contenuti nel secondo `Iterable` (seconda fase). Precisamente, appena l' i -esimo `Runnable` del primo gruppo termina, quel thread passa ad eseguire l' i -esimo `Runnable` del secondo gruppo.

449. (StringQuiz, 2015-9-21)

La classe `StringQuiz` consente a diversi thread di tentare di indovinare una stringa segreta, entro un tempo prestabilito. Il costruttore accetta la stringa segreta e un timeout in millisecondi. Il metodo `guess` accetta una stringa e restituisce vero se è uguale a quella segreta e falso altrimenti. Ciascun thread ha 3 possibilità di indovinare, oltre le quali il metodo `guess` lancia un'eccezione. Scaduto il timeout, il metodo `guess` lancia un'eccezione ogni volta che viene invocato. La classe deve risultare *thread-safe*.

450. (TimeToFinish, 2015-7-8)

Implementare la classe thread-safe `TimeToFinish`, che permette a diversi thread di comunicare quanto tempo manca alla propria terminazione. Il metodo `setEstimate` accetta un long che rappresenta il numero di millisecondi che mancano al thread chiamante per terminare la sua esecuzione (cioè, il *time-to-finish*). Il metodo `maxTimeToFinish` restituisce *in tempo costante* il numero di millisecondi necessari perché tutti i thread terminino. La stringa restituita da `toString` riporta il time-to-finish di tutti i thread.

Suggerimento: il metodo statico `System.currentTimeMillis()` restituisce un long che rappresenta il numero di millisecondi trascorsi dal 1 gennaio 1970 (*POSIX time*).

Caso d'uso:

```
TimeToFinish ttf = new TimeToFinish();
ttf.setEstimate(5000);
// a questo punto un altro thread invoca ttf.setEstimate(3000)
Thread.sleep(500);
System.out.println("Tempo rimanente: " + ttf.maxTimeToFinish() + " millisecondi.");
System.out.println(ttf);
```

Output:

```
Tempo rimanente: 4500 millisecondi.
Thread 1: 2500
Thread main: 4500
```

451. (SimpleThread, 2015-6-24)

Indicare tutti gli output possibili di un programma che faccia partire contemporaneamente due istanze della seguente classe `SimpleThread`.

```
public class SimpleThread extends Thread
{
    private static volatile int n = 0;

    public void run() {
        n++;
        int m = n;
        System.out.println(m);
    }
}
```

452. (ForgetfulSet, 2015-2-5)

Realizzare la classe `ForgetfulSet`, che rappresenta un insieme che “dimentica” progressivamente gli oggetti inseriti. Precisamente, ciascun oggetto inserito viene rimosso automaticamente dopo un ritardo specificato inizialmente come parametro del costruttore. Quindi, il costruttore accetta il ritardo in millisecondi; il metodo `add` inserisce un oggetto nell’insieme; il metodo `contains` accetta un oggetto e restituisce `true` se e solo se quell’oggetto appartiene correntemente all’insieme.

La classe deve utilizzare un numero limitato di thread e deve risultare *thread-safe*.

Suggerimento: il metodo statico `System.currentTimeMillis()` restituisce il numero di millisecondi trascorsi dal 1 gennaio 1970 (*POSIX time*).

Esempio d'uso:

```
ForgetfulSet<String> s = new ForgetfulSet<String>(1000);
s.add("uno");
s.add("due");
System.out.println(s.contains("uno") + ", " + s.contains("due") + ", " + s.contains("tre"));
Thread.sleep(800);
s.add("tre");
System.out.println(s.contains("uno") + ", " + s.contains("due") + ", " + s.contains("tre"));
Thread.sleep(800);
System.out.println(s.contains("uno") + ", " + s.contains("due") + ", " + s.contains("tre"));
```

Output:

```
true, true, false
true, true, true
false, false, true
```

453. (atLeastOne, 2014-9-18)

Implementare il metodo statico `atLeastOne`, che accetta come argomenti un intero positivo n e un `Runnable r` ed esegue in parallelo n copie di `r`. Appena una delle copie termina, le altre vengono interrotte (con `interrupt`) e il metodo restituisce il controllo al chiamante.

454. (Exchanger, 2014-7-3)

Un `Exchanger` è un oggetto che serve a due thread per scambiarsi due oggetti dello stesso tipo. Ciascun thread invocherà il metodo `exchange` passandogli il proprio oggetto e riceverà come risultato l'oggetto passato dall'altro thread. Il primo thread che invoca `exchange` dovrà aspettare che anche il secondo thread invochi quel metodo, prima di ricevere il risultato. Quindi, il metodo `exchange` risulta bloccante per il primo thread che lo invoca e non bloccante per il secondo.

Un `Exchanger` può essere usato per un solo scambio. Ulteriori chiamate ad `exchange` dopo le prime due portano ad un'eccezione.

Nell'esempio d'uso, due thread condividono il seguente oggetto:

| | |
|--|---|
| <code>Exchanger<String> e = new Exchanger<String>();</code> | |
| Thread 1: | Thread 2: |
| <code>String a = e.exchange("ciao");</code> <code>System.out.println(a);</code> | <code>String a = e.exchange("Pippo");</code> <code>System.out.println(a);</code> |
| Output thread 1: <code>Pippo</code> | Output thread 2: <code>ciao</code> |

455. (PriorityExecutor, 2014-7-28)

Un `PriorityExecutor` è un thread che esegue in sequenza una serie di task, in ordine di priorità. Il metodo `addTask` accetta un `Runnable` e una priorità (numero intero) e lo aggiunge ad una coda di task. Quando il `PriorityExecutor` è libero (cioè, non sta eseguendo alcun task), preleva dalla coda *uno dei task con priorità massima* e lo esegue.

Sono preferibili le implementazioni in cui né `addTask` né la ricerca del prossimo task da eseguire richiedano tempo lineare.

| | |
|--|---|
| Esempio d'uso: | Output: |
| <pre>Runnable r1 = ..., r2 = ...; PriorityExecutor e = new PriorityExecutor(); e.addTask(r2, 10); e.addTask(r1, 100); e.start(); e.addTask(r2, 15); e.addTask(r1, 50);</pre> | prima viene eseguito due volte il task r1, poi due volte il task r2 |

456. (PeriodicTask, 2014-3-5)

Realizzare la classe `PeriodicTask`, che consente di eseguire un `Runnable` periodicamente, ad intervalli specificati. Il costruttore accetta un oggetto `Runnable` *r* e un numero di millisecondi *p*, detto *periodo*, e fa partire un thread che esegue *r.run()* ogni *p* millisecondi (si noti che il costruttore non è bloccante). Il metodo `getTotalTime` restituisce il numero complessivo di millisecondi che tutte le chiamate a *r.run()* hanno utilizzato fino a quel momento.

Suggerimento: il seguente metodo della classe `System` restituisce il numero di millisecondi trascorsi dal primo gennaio 1970: `public static long currentTimeMillis()`.

(15 punti) Inoltre, dire quali dei seguenti criteri di uguaglianza per oggetti di tipo `PeriodicTask` sono validi, giustificando brevemente la risposta. Due oggetti di tipo `PeriodicTask` sono uguali se:

- a) hanno lo stesso `Runnable` ed un periodo inferiore ad un secondo;
- b) hanno due periodi che sono l'uno un multiplo intero dell'altro (ad es. 5000 millisecondi e 2500 millisecondi);
- c) hanno lo stesso `Runnable` oppure lo stesso periodo.

| | |
|---|--|
| Esempio d'uso: | Output: |
| <pre>Runnable r = new Runnable() { public void run() { System.out.println("Ciao!"); } }; new PeriodicTask(r, 2000);</pre> | Ciao! Ciao! (dopo 2 secondi) Ciao! (dopo altri 2 secondi) ... |

457. (Alarm, 2014-11-28)

Realizzare la classe `Alarm`, nel contesto di un sistema di allarme domestico. Il compito dell'oggetto `Alarm` è di stampare un messaggio se una condizione anomala permane oltre una soglia di tempo prestabilita (un *timeout*). Il costruttore accetta il timeout in secondi. Il metodo `anomalyStart` segnala l'inizio di una situazione anomala, mentre il metodo `anomalyEnd` ne segnala la fine. Se viene invocato `anomalyStart` e poi non viene invocato `anomalyEnd` entro il timeout, l'oggetto `Alarm` produce in output il messaggio "Allarme!".

Se `anomalyStart` viene invocato più volte, senza che sia ancora stato invocato `anomalyEnd`, le invocazioni successive alla prima vengono ignorate (cioè, non azzerano il timeout).

La classe `Alarm` deve risultare *thread safe* e i suoi metodi non devono essere bloccanti.

| | |
|--|--------------------------|
| Esempio d'uso: | Output: (dopo 5 secondi) |
| <pre>Alarm a = new Alarm(5); a.anomalyStart();</pre> | Allarme! |

458. (PostOfficeQueue, 2014-1-31)

Implementare una classe `PostOfficeQueue`, che aiuta a gestire la coda in un ufficio postale. Il costruttore accetta il numero totale di sportelli disponibili. Quando l'*i*-esimo sportello comincia a servire un cliente e quindi diventa occupato, viene invocato (dall'esterno della classe) il metodo `deskStart` passando *i* come argomento. Quando invece l'*i*-esimo sportello si libera, viene invocato il metodo `deskFinish` con argomento *i*. Infine, il metodo `getFreeDesk` restituisce il numero di uno sportello libero. Se non ci sono sportelli liberi, il metodo attende che se ne liberi uno.

Si deve supporre che thread diversi possano invocare concorrentemente i metodi di `PostOfficeQueue`. È necessario evitare *race condition* ed attese attive.

| | |
|---|------------------|
| Esempio d'uso: | Output: |
| <pre>PostOfficeQueue poq = new PostOfficeQueue(5); System.out.println(poq.getFreeDesk()); poq.deskStart(0); System.out.println(poq.getFreeDesk()); poq.deskStart(1); poq.deskStart(2); System.out.println(poq.getFreeDesk());</pre> | <pre>0 1 3</pre> |

459. (executeWithDeadline, 2013-9-25)

Implementare il metodo statico `executeWithDeadline`, che accetta un riferimento *r* a un `Runnable` ed un tempo *t* espresso in secondi. Il metodo esegue *r* fino ad un tempo massimo di *t* secondi, trascorsi i quali il metodo interromperà *r* e restituirà il controllo al chiamante.

Quindi, il metodo deve terminare quando una delle seguenti condizioni diventa vera: 1) il `Runnable` termina la sua esecuzione, oppure 2) trascorrono *t* secondi.

Si può supporre che il `Runnable` termini quando viene interrotto.

460. (processArray, 2013-7-9)

L'interfaccia `RunnableFunction` rappresenta una funzione che accetta un parametro e restituisce un valore dello stesso tipo.

```
interface RunnableFunction<T> {
    public T run(T x);
}
```

Implementare il metodo statico `processArray`, che esegue una data `RunnableFunction` *f* su tutti gli elementi di un array di input e memorizza i risultati in un altro array, di output. Inoltre, il metodo riceve come argomento un intero *n* ed utilizza *n* thread diversi per eseguire la funzione *f* contemporaneamente su diversi elementi dell'array.

Ad esempio, se n = 2, il metodo potrebbe lanciare due thread che eseguono f sui primi due elementi dell'array. Poi, appena uno dei due thread termina, il metodo potrebbe lanciare un nuovo thread che esegue f sul terzo elemento dell'array, e così via.

Rispondere alla seguente domanda: nella vostra implementazione, quand'è che il metodo `processArray` restituisce il controllo al chiamante?

| | |
|---|---------------------------------|
| Esempio d'uso: | Output: |
| <pre>String[] x = {"uno", "due", "tre"}, y = new String[3]; RunnableFunction<String> f = new RunnableFunction<String>() { public String run(String x) { return x + x; } }; processArray(x, y, f, 2); for (String s: y) System.out.println(s);</pre> | <pre>unouno duedue tretre</pre> |

461. (MultiBuffer, 2013-6-25)

Implementare la classe parametrica `MultiBuffer`, che rappresenta un insieme di buffer. Il suo costruttore accetta il numero *n* di buffer da creare. Il metodo `insert` inserisce un oggetto nel buffer che in quel momento contiene meno elementi. Il metodo bloccante `take` accetta un intero *i* compreso tra 0 ed *n* – 1 e restituisce il primo oggetto presente nel buffer *i*-esimo. La classe deve risultare *thread-safe*.

| | |
|--|---------------|
| Esempio d'uso: | Output: 13 |
| <pre>MultiBuffer<Integer> mb = new MultiBuffer<Integer>(3); mb.insert(13); mb.insert(24); mb.insert(35); System.out.println(mb.take(0));</pre> | |

Si consideri il seguente schema di sincronizzazione: `insert` è mutuamente esclusivo con `take(i)`, per ogni valore di `i`; `take(i)` è mutuamente esclusivo con `take(i)`, ma è compatibile con `take(j)`, quando `j` è diverso da `i`. Rispondere alle seguenti domande:

- a) Questo schema evita le *race condition*?
- b) E' possibile implementare questo schema utilizzando metodi e blocchi sincronizzati?

462. (Shared average, 2013-3-22)

La classe `SharedAverage` permette a diversi thread di comunicare un valore numerico (double) e poi ricevere il valore medio tra tutti quelli inviati dai diversi thread. Precisamente, il costruttore accetta come argomento il numero `n` di thread che parteciperà all'operazione. Il metodo `sendValAndReceiveAvg` accetta come argomento il valore indicato dal thread corrente, mette il thread corrente in attesa che tutti gli `n` thread abbiano inviato un valore, e infine restituisce la media aritmetica di tutti i valori inviati.

Se un thread ha già chiamato `sendValAndReceiveAvg` una volta, al secondo tentativo viene sollevata un'eccezione.

È necessario evitare *race condition*.

| | |
|---|---|
| Esempio d'uso: | Comportamento: Quando altri 9 thread avranno invocato <code>sendValAndReceiveAvg</code> , tutte le invocazioni restituiranno la media dei 10 valori inviati. |
| <pre>SharedAverage a = new SharedAverage(10); double average; try { average = a.sendValAndReceiveAvg(5.0); } catch (InterruptedException e) { return; }</pre> | |

463. (Concurrent filter, 2013-2-11)

Data la seguente interfaccia:

```
public interface Selector<T> {
    boolean select(T x);
}
```

implementare il metodo (statico) `concurrentFilter`, che prende come argomenti un `Set X` e un `Selector S`, di tipi compatibili, e restituisce un nuovo insieme `Y` che contiene quegli elementi di `X` per i quali la funzione `select` di `S` restituisce il valore `true`.

Inoltre, il metodo deve invocare la funzione `select` in parallelo su tutti gli elementi di `X` (dovrà quindi creare tanti thread quanti sono gli elementi di `X`).

| | |
|---|-------------------|
| Esempio d'uso: | Output: 1 5 |
| <pre>Set<Integer> x = new HashSet<Integer>(); x.add(1); x.add(2); x.add(5); Selector<Integer> oddSelector = new Selector<Integer>() { public boolean select(Integer n) { return (n%2 != 0); } }; Set<Integer> y = concurrentFilter(x, oddSelector); for (Integer n: y) System.out.println(n);</pre> | |

464. (concurrentMax, 2013-12-16)

Implementare il metodo statico `concurrentMax`, che accetta un riferimento ad una matrice di interi e restituisce il massimo valore presente nella matrice. Internamente, il metodo crea tanti thread quante sono le righe della matrice. Ciascuno di questi thread ricerca il massimo all'interno della sua riga e poi aggiorna il massimo globale.

È necessario evitare *race condition* ed attese attive.

| Esempio d'uso: | Output: |
|---|---------|
| <pre>int [][] arr = { {23, 23, 45, 2, 4}, {-10, 323, 33, 445, 4}, {12, 44, 90, 232, 122} }; System.out.println(concurrentMax(arr));</pre> | 445 |

465. (Shared object, 2013-1-22)

Elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) throws InterruptedException {
    class MyThread extends Thread {
        private int id;
        private int[] arr;
        public MyThread(int id, int[] arr) {
            this.id = id;
            this.arr = arr;
        }
        public void run() {
            synchronized (arr) {
                arr[0]++;
                System.out.println(id + ":" + arr[0]);
            }
            return;
        }
    }
    int[] a = { 0 };
    Thread t1 = new MyThread(1,a);
    Thread t2 = new MyThread(2,a);
    Thread t3 = new MyThread(3,a);
    t1.start(); t2.start(); t3.start();
}
```

466. (Mystery thread 2, 2012-9-3)

Elencare tutte le sequenze di output possibili per il seguente programma.

```
public static void main(String[] args) {
    class MyThread extends Thread {
        private int id;
        private Thread other;
        public MyThread(int n, Thread t) {
            id = n;
            other = t;
        }
        public void run() {
            try {
                if (other!=null)
                    other.join();
            } catch (InterruptedException e) {
                return;
            }
            System.out.println(id);
        }
    }
}
```

```

    }
    Thread t1 = new MyThread(1,null);
    Thread t2 = new MyThread(2,null);
    Thread t3 = new MyThread(3,t1);
    Thread t4 = new MyThread(4,t2);
    t1.start(); t2.start(); t3.start(); t4.start();
}
}

```

467. (Mystery thread, 2012-7-9)

Escludendo i cosiddetti *spurious wakeup* (risvegli da `wait` senza che sia avvenuta una `notify`), elencare tutte le sequenze di output possibili per il seguente programma.

```

public static void main(String[] args) throws InterruptedException {
    final Object dummy = new Object();
    final Thread t1 = new Thread() {
        public void run() {
            synchronized (dummy) {
                while (true) {
                    try { dummy.wait(); System.out.println("T1:A"); }
                    catch (InterruptedException e) { break; }
                }
                System.out.println("T1:B");
            }
        }
    };
    final Thread t2 = new Thread() {
        public void run() {
            synchronized (dummy) {
                dummy.notifyAll();
                System.out.println("T2:A");
            }
            t1.interrupt();
            System.out.println("T2:B");
        }
    };
    t1.start();
    t2.start();
    t1.join();
    System.out.println("Fine");
}

```

468. (ThreadRace, 2012-6-18)

Implementare il metodo statico `threadRace`, che accetta due oggetti `Runnable` come argomenti, li esegue contemporaneamente e restituisce 1 oppure 2, a seconda di quale dei due `Runnable` è terminato prima.

Si noti che `threadRace` è un metodo bloccante. Sarà valutato negativamente l'uso di attesa attiva.

469. (MultiProgressBar, 2011-3-4)

Si supponga che una applicazione divida un'operazione complessa tra più thread, che procedono in parallelo. Si implementi la classe `MultiProgressBar`, di cui ciascun oggetto serve a tenere traccia dello stato di avanzamento dei thread coinvolti in una data operazione.

Il costruttore accetta il numero totale n di thread coinvolti. Il metodo `progress`, con un argomento intero e senza valore di ritorno, consente ad un thread di dichiarare il suo stato di avanzamento, in percentuale. Tale metodo lancia un'eccezione se lo stesso thread dichiara uno stato di avanzamento inferiore ad uno precedentemente dichiarato. Il metodo `getProgress`, senza argomenti e con valore di ritorno intero, restituisce il *minimo* stato di avanzamento tra tutti i thread coinvolti.

E' necessario evitare eventuali *race condition*.

Un esempio d'uso verrà fornito alla lavagna.

470. (VoteBox, 2011-2-7)

Si implementi la classe `VoteBox`, che rappresenta un'urna elettorale, tramite la quale diversi thread possono votare tra due alternative, rappresentate dai due valori booleani. Il costruttore accetta il numero totale n di thread aventi diritto al voto. La votazione termina quando n thread diversi hanno votato. In caso di pareggio, vince il valore false.

Il metodo `vote`, con parametro boolean e nessun valore di ritorno, permette ad un thread di votare, e solleva un'eccezione se lo stesso thread tenta di votare più di una volta. Il metodo `waitForResult`, senza argomenti e con valore di ritorno booleano, restituisce il risultato della votazione, mettendo il thread corrente in attesa se la votazione non è ancora terminata. Infine, il metodo `isDone` restituisce true se la votazione è terminata, e false altrimenti.

E' necessario evitare eventuali *race condition*.

| Esempio d'uso: | Output dell'esempio d'uso: |
|--|--|
| <pre>VoteBox b = new VoteBox(10); b.vote(true); System.out.println(b.isDone()); b.vote(false);</pre> | <pre>false Exception in thread "main"...</pre> |

471. (ExecuteInParallel, 2010-9-14)

Implementare il metodo statico `executeInParallel`, che accetta come argomenti un array di `Runnable` e un numero naturale k , ed esegue tutti i `Runnable` dell'array, k alla volta.

In altre parole, all'inizio il metodo fa partire i primi k `Runnable` dell'array. Poi, non appena uno dei `Runnable` in esecuzione termina, il metodo ne fa partire un altro, preso dall'array, fino ad esaurire tutto l'array.

Sarà valutato negativamente l'uso di attesa attiva.

472. (QueueOfTasks, 2010-6-28)

Implementare la classe `QueueOfTasks`, che rappresenta una sequenza di azioni da compiere, ciascuna delle quali rappresentata da un oggetto `Runnable`. Il metodo `add` aggiunge un'azione alla sequenza. Le azioni vengono eseguite nell'ordine in cui sono state passate ad `add` (FIFO), una dopo l'altra, automaticamente.

| Esempio d'uso: | Output dell'esempio d'uso: |
|---|---|
| <pre>QueueOfTasks q = new QueueOfTasks(); Runnable r1 = new Runnable() { public void run() { try { Thread.sleep(2000); } catch (InterruptedException e) { return; } System.out.println("Io_adoro_i_thread!"); } }; Runnable r2 = new Runnable() { public void run() { System.out.println("Io_odo_i_thread!"); } }; q.add(r1); q.add(r1); q.add(r2); System.out.println("fatto.");</pre> | <pre>fatto. Io adoro i thread! (dopo 2 secondi) Io adoro i thread! (dopo 2 secondi) Io odio i thread!</pre> |

473. (Auction, 2009-9-1'8)

La classe `Auction` rappresenta una vendita all'asta. Il suo costruttore accetta come argomento il prezzo di partenza dell'asta. Il metodo `makeOffer` rappresenta la presentazione di un'offerta e prende come argomenti l'ammontare dell'offerta e il nome dell'acquirente.

Un oggetto `Auction` deve accettare offerte, finchè non riceve offerte per 3 secondi consecutivi. A quel punto, l'oggetto stampa a video l'offerta più alta e il nome del compratore.

Si supponga che più thread possano chiamare concorrentemente il metodo `makeOffer` dello stesso oggetto.

| | |
|--|--|
| Esempio d'uso: | Output dell'esempio d'uso: Oggetto venduto a Giulia per 1500 euro. |
| <pre>Auction a = new Auction(1000); a.makeOffer(1100, "Marco"); a.makeOffer(1200, "Luca"); Thread.sleep(1000); a.makeOffer(200, "Anna"); Thread.sleep(1000); a.makeOffer(1500, "Giulia'"); Thread.sleep(4000);</pre> | |

474. (**Elevator, 2009-7-9**)

Implementare la classe `Elevator`, che simula il comportamento di un ascensore. Il costruttore prende come argomento il numero di piani serviti (oltre al pian terreno). Il metodo `call` rappresenta la prenotazione ("chiamata") di un piano. Se l'argomento di `call` è fuori dall'intervallo corretto, viene lanciata un'eccezione.

In un thread indipendente, quando ci sono chiamate in attesa, l'ascensore cambia piano in modo da soddisfare una delle chiamate, scelta in ordine arbitrario. L'ascensore impiega due secondi per percorrere ciascun piano e stampa a video dei messaggi esplicativi, come nel seguente caso d'uso.

Attenzione: verrà valutato negativamente l'uso di attesa attiva.

| | |
|---|---|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <code>Elevator e = new Elevator(10);</code> | <code>Elevator leaves floor 0</code> |
| <code>e.call(8);</code> | <code>Elevator stops at floor 3</code> (dopo 6 secondi) |
| <code>e.call(5);</code> | <code>Elevator leaves floor 3</code> |
| <code>e.call(3);</code> | <code>Elevator stops at floor 4</code> (dopo 2 secondi) |
| <code>e.call(4);</code> | <code>Elevator leaves floor 4</code> |
| | <code>Elevator stops at floor 5</code> (dopo 2 secondi) |
| | <code>Elevator leaves floor 5</code> |
| | <code>Elevator stops at floor 8</code> (dopo 6 secondi) |

475. (**RunnableWithArg, 2008-9-8**)

Si consideri la seguente interfaccia.

```
public interface RunnableWithArg<T> {
    void run(T x);
}
```

Un oggetto `RunnableWithArg` è simile ad un oggetto `Runnable`, tranne per il fatto che il suo metodo `run` accetta un argomento.

Si implementi una classe `RunOnSet` che esegue il metodo `run` di un oggetto `RunnableWithArg` su tutti gli oggetti di un dato insieme, *in parallelo*.

| | |
|--|--|
| <p>Esempio d'uso:</p> <pre>Set<Integer> s = new HashSet<Integer>(); s.add(3); s.add(13); s.add(88); RunnableWithArg<Integer> r = new RunnableWithArg<Integer>() { public void run(Integer i) { System.out.println(i/2); } }; Thread t = new RunOnSet<Integer>(r, s); t.start();</pre> | <p>Un possibile output dell'esempio d'uso:</p> <pre>1 6 44</pre> |
|--|--|

476. (**MutexWithLog**, 2008-7-9)

Implementare la classe **MutexWithLog** che rappresenta un mutex, con i classici metodi **lock** e **unlock**, che in aggiunta scrive un messaggio a video ogni volta che un thread riesce ad acquisirlo e ogni volta che un thread lo rilascia. Il metodo **unlock** deve lanciare un'eccezione se viene chiamato da un thread diverso da quello che ha acquisito il mutex.

| | |
|---|---|
| <p>Esempio d'uso:</p> <pre>final MutexWithLog m = new MutexWithLog(); Thread t = new Thread("Secondo") { public void run() { m.lock(); System.out.println("Due!"); m.unlock(); } }; t.start(); m.lock(); System.out.println("Uno!"); m.unlock();</pre> | <p>Un possibile output dell'esempio d'uso:</p> <pre>"main" ha acquisito il lock Uno! "main" ha rilasciato il lock "Secondo" ha acquisito il lock Due! "Secondo" ha rilasciato il lock</pre> |
|---|---|

477. (**RunnableWithProgress**, 2008-6-19)

Si consideri la seguente interfaccia.

```
public interface RunnableWithProgress extends Runnable {
    int getProgress();
}
```

Un oggetto **RunnableWithProgress** rappresenta un oggetto **Runnable**, che in più dispone di un metodo che restituisce la percentuale di lavoro completata dal metodo **run** fino a quel momento.

Si implementi una classe **ThreadWithProgress** che esegua un oggetto **RunnableWithProgress** mostrando ad intervalli regolari la percentuale di lavoro svolto fino a quel momento. Precisamente, **ThreadWithProgress** deve stampare a video ogni secondo la percentuale di lavoro aggiornata, a meno che la percentuale non sia la stessa del secondo precedente, nel qual caso la stampa viene saltata.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre>RunnableWithProgress r = new RunnableWithProgress() {...}; Thread t = new ThreadWithProgress(r); t.start();</pre> | <p>Un possibile output dell'esempio d'uso:</p> <pre>5% 12% 25% 70% 90% 100%</pre> |
|--|---|

478. (DelayIterator, 2008-3-27)

Implementare un metodo statico `delayIterator` che prende come argomenti un iteratore `i` ed un numero intero `n`, e restituisce un nuovo iteratore dello stesso tipo di `i`, che restituisce gli stessi elementi di `i`, ma in cui ogni elemento viene restituito (dal metodo `next`) dopo un ritardo di `n` secondi. Viene valutato positivamente l'uso di classi anonime.

Si ricordi che nella classe `Thread` è presente il metodo:

```
public static void sleep(long milliseconds) throws InterruptedException
```

479. (Simulazione di ParkingLot, 2007-7-20)

Utilizzando la classe `ParkingLot` descritta nell'esercizio 3, scrivere un programma che simula l'ingresso e l'uscita di veicoli da un parcheggio. Un primo thread aggiunge un veicolo ogni secondo (a meno che il parcheggio non sia pieno). Un secondo thread, ogni due secondi, rimuove un veicolo dal parcheggio (a meno che il parcheggio non sia vuoto) e stampa a video il numero di secondi che tale veicolo ha trascorso nel parcheggio. Non ha importanza in che ordine i veicoli vengono rimossi dal parcheggio.

480. (Highway, 2007-6-29)

Implementare una classe `Highway`, che rappresenti un'autostrada a senso unico. Il costruttore accetta la lunghezza dell'autostrada in chilometri. Il metodo `insertCar` prende un intero `x` come argomento ed aggiunge un'automobile al chilometro `x`. L'automobile inserita percorrerà l'autostrada alla velocità di un chilometro al minuto, (60 km/h) fino alla fine della stessa. Il metodo `nCars` prende un intero `x` e restituisce il numero di automobili presenti al chilometro `x`. Il metodo `progress` simula il passaggio di 1 minuto di tempo (cioè fa avanzare tutte le automobili di un chilometro).

Si supponga che thread multipli possano accedere allo stesso oggetto `Highway`.

Dei 25 punti, 8 sono riservati a coloro che implementeranno `progress` in tempo indipendente dal numero di automobili presenti sull'autostrada.

Esempio d'uso:

```
Highway h = new Highway(10);
h.insertCar(3); h.insertCar(3); h.insertCar(5);

System.out.println(h.nCars(4));
h.progress();
System.out.println(h.nCars(4));
```

Output:

```
0
2
```

481. (2006-9-15)

Individuare e descrivere sinteticamente gli eventuali errori nel seguente programma. Il programma dovrebbe lanciare un nuovo thread che stampa gli interi da 0 a 9.

```
1 class Test extends Runnable {
2     private Thread thread;
3
4     public Test() {
5         thread = new Thread();
6     }
7
8     public run() {
9         int i = 0;
10        for (i=0; i<10 ;i++)
11            System.out.println("i=" + i);
12    }
13
14    public static void main(String args[]) {
15        Test t = new Test();
16        t.start();
```

```
17 }  
18 }
```

17 Iteratori e ciclo for-each

482. (FunnyIterator, 2014-11-3)

Individuare l'output del seguente programma. Dire se la classe `FunnyIterator` rispetta il contratto dell'interfaccia `Iterator`. In caso negativo, giustificare la risposta.

```
1 public class FunnyIterator implements Iterator {  
2     private String msg = "";  
3  
4     public Object next() {  
5         if (msg.equals("")) msg = "ah";  
6         else msg += msg;  
7         return msg;  
8     }  
9  
10    public boolean hasNext() { return msg.length() < 5; }  
11    public void remove() {}  
12    public void addChar() { msg += "b"; }  
13  
14    public static void main(String[] args) {  
15        Iterator i = new FunnyIterator();  
16  
17        do {  
18            System.out.println(i.next());  
19        } while (i.hasNext());  
20    }  
21 }
```

483. (IncreasingSubsequence, 2009-9-1'8)

Implementare la classe `IncreasingSubseq` che, data una lista di oggetti tra loro confrontabili, rappresenta la *sottosequenza crescente* che inizia col primo elemento.

Attenzione: la classe deve funzionare con qualunque tipo di dato che sia confrontabile (non solo con “Integer”).

Sarà valutato negativamente l'uso di “strutture di appoggio”, ovvero di spazio aggiuntivo di dimensione non costante.

Esempio d'uso:

```
List<Integer> l = new LinkedList<Integer>();  
l.add(10); l.add(3);  
l.add(5); l.add(12);  
l.add(11); l.add(35);  
for (Integer i: new IncreasingSubseq<Integer>(l))  
    System.out.println(i);
```

Output dell'esempio d'uso:

```
10  
12  
35
```

484. (CrazyIterator, 2008-4-21)

Individuare l'output del seguente programma. Dire se la classe `CrazyIterator` rispetta il contratto dell'interfaccia `Iterator`. In caso negativo, giustificare la risposta.

```
1 public class CrazyIterator implements Iterator {  
2     private int n = 0;  
3  
4     public Object next() {  
5         int j;  
6         while (true) {  
7             for (j=2; j<=n/2; j++) if (n % j == 0) break;
```

```

8      if (j > n/2) break;
9      else n++;
10     }
11     return new Integer(n);
12   }
13
14   public boolean hasNext() { n++; return true; }
15   public void remove() { throw new RuntimeException(); }
16
17   public static void main(String[] args) {
18     Iterator i = new CrazyIterator();
19
20     while (i.hasNext() && (Integer)i.next()<10) {
21       System.out.println(i.next());
22     }
23   }
24 }
```

485. (**MyFor**, 2008-2-25)

Implementare una classe **MyFor** in modo che, per tutti i numeri interi *a*, *b* e *c*, il ciclo:

```
for (Integer i: new MyFor(a, b, c)) { ... }
```

sia equivalente al ciclo:

```
for (Integer i=a; i<b ; i+=c) { ... }
```

486. (**Selector**, 2007-9-17)

L'interfaccia parametrica **Selector** prevede un metodo **select** che restituisce un valore booleano per ogni elemento del tipo parametrico.

```
public interface Selector<T> {
  boolean select(T x);
}
```

Implementare una classe **SelectorIterator** che accetta una collezione e un selettore dello stesso tipo, e permette di iterare sugli elementi della collezione per i quali il selettore restituisce **true**.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre> Integer [] a = { 1, 2, 45, 56, 343, 22, 12, 7, 56}; List<Integer> l = Arrays.asList(a); Selector<Integer> pari = new Selector<Integer>() { public boolean select(Integer n) { return (n % 2) == 0; } }; for (Integer n: new SelectorIterator<Integer>(l, pari)) System.out.print(n + " "); </pre> | 2 56 22 12 56 |

487. (**CommonDividers**, 2007-7-20)

Implementare una classe **CommonDividers** che rappresenta tutti i divisori comuni di due numeri interi, forniti al costruttore. Su tale classe si deve poter iterare secondo il seguente caso d'uso. Dei 30 punti, 15 sono riservati a coloro che realizzeranno l'iteratore senza usare spazio aggiuntivo. Viene valutato positivamente l'uso di classi anonime laddove opportuno.

| | |
|---|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre> for (Integer n: new CommonDividers(12, 60)) System.out.print(n + " "); </pre> | 1 2 3 4 6 12 |

488. (AncestorIterator, 2007-4-26)

Con riferimento all'Esercizio 1, definire una classe `AncestorIterator` che itera su tutti gli antenati conosciuti di una persona, in ordine arbitrario. Ad esempio, si consideri il seguente caso d'uso, che fa riferimento alle persone a,b,c,d ed e dell'Esercizio 1.

| | |
|--|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Iterator i = new AncestorIterator(e); while (i.hasNext()) { System.out.println(i.next()); }</pre> | Luca Rossi Mario Rossi Luisa Verdi |

Dei 25 punti, 10 sono riservati a coloro che implementaranno `AncestorIterator` come classe interna di `Person`. In tal caso, il primo rigo dell'esempio d'uso diventa:

```
Iterator i = e.new AncestorIterator();
```

Suggerimento: si ricorda che se B è una classe interna di A, all'interno di B il riferimento implicito all'oggetto di tipo A si chiama `A.this`.

489. (Primes, 2007-2-23)

Definire una classe `Primes` che rappresenta l'insieme dei numeri primi. Il campo statico `iterable` fornisce un oggetto su cui si può iterare, ottenendo l'elenco di tutti i numeri primi. Non deve essere possibile per un'altra classe creare oggetti di tipo `Primes`.

Suggerimento: `Primes` potrebbe implementare sia `Iterator<Integer>` che `Iterable<Integer>`. In tal caso, il campo `iterable` potrebbe puntare ad un oggetto di tipo `Primes`.

| | |
|--|--|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>for (Integer i: Primes.iterable) { if (i > 20) break; System.out.println(i); }</pre> | 1 3 5 7 11 13 17 19 |

490. (SuperclassIterator, 2006-9-15)

Implementare una classe `SuperclassIterator` che rappresenta un iteratore su tutte le superclassi di un oggetto dato, a partire dalla classe stessa dell'oggetto fino ad arrivare ad `Object`.

Ad esempio, nell'ambito della tradizionale gerarchia formata dalle classi `Employee` e `Manager`, si consideri il seguente caso d'uso.

| | |
|---|---|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Iterator<Class<?>> i = new SuperclassIterator(new Manager("Franco")); while (i.hasNext()) System.out.println(i.next());</pre> | class Manager class Employee class java.lang.Object |

491. (TwoSteps, 2006-7-17)

Implementare un metodo statico `twoSteps` che accetta come argomento un iteratore e restituisce un iteratore dello stesso tipo, che compie due passi per ogni chiamata a `next`.

Come esempio, si consideri il seguente caso d'uso.

Esempio d'uso:

```
List<Integer> l = new LinkedList<Integer>();
l.add(3); l.add(5); l.add(7); l.add(9);

Iterator<Integer> iter1 = twoSteps(l.iterator());
System.out.println("Iterazione_1:");
System.out.println(iter1.next());
System.out.println(iter1.next());

Iterator<Integer> iter2 = twoSteps(l.iterator());
System.out.println("Iterazione_2:");
while (iter2.hasNext())
    System.out.println(iter2.next());
```

Output dell'esempio d'uso:

```
Iterazione 1:
3
7
Iterazione 2:
3
7
```

492. (BinaryTreePreIterator, 2006-4-27)

Il seguente frammento di classe definisce un nodo in un albero binario.

```
public class BinaryTreeNode {
    private BinaryTreeNode left, right;
    public BinaryTreeNode getLeft() { return left; }
    public BinaryTreeNode getRight() { return right; }
}
```

Si implementi una classe iteratore `BinaryTreePreIterator` che visiti i nodi dell'albero in preorder (ciascun nodo prima dei suoi figli). Tale classe deve poter essere usata nel seguente modo:

```
public static void main(String[] args) {
    BinaryTreeNode root = ....;
    Iterator i = new BinaryTreePreIterator(root);
    while (i.hasNext()) {
        BinaryTreeNode node = (BinaryTreeNode) i.next();
        ...
    }
```

18 Criteri di ordinamento tra oggetti

493. (Rotating list comparator, 2019-4-29)

Dire quali dei seguenti sono criteri validi per un comparatore tra oggetti RotatingList dell'esercizio precedente. In caso di validità, dire anche se il criterio è coerente con la definizione di equals dell'esercizio precedente.

compare(x, y) restituisce (nei casi non contemplati, il metodo restituisce 0):

- a) -1 se le due liste hanno lunghezza almeno 3 e x si può trasformare in y tramite una singola rotazione a destra; 1 se le due liste hanno lunghezza almeno 3 e x si può trasformare in y tramite una singola rotazione a sinistra.
- b) -1 se x è un prefisso proprio di y (come "casa" è un prefisso proprio di "casata"); 1 se y è un prefisso proprio di x .
- c) -1 se y contiene un oggetto che non è presente in x ; 1 se x contiene tutti gli oggetti di y , e anche un oggetto non presente in y .
- d) -1 se x ha lunghezza pari e y dispari; 1 se y ha lunghezza pari e x dispari.

494. (Date, 2018-6-20)

La classe Date rappresenta una data tramite tre numeri interi (giorno, mese e anno) e ridefinisce equals nel modo naturale.

Dire quali delle seguenti sono specifiche valide per un comparatore tra due oggetti Date a e b . Dire anche quali specifiche sono coerenti con equals.

compare(a, b) restituisce (nei casi non contemplati, restituisce 0):

- a) -1 se l'anno di a è minore di quello di b ; 1 se l'anno di a è maggiore di quello di b .
- b) -1 se a e b hanno lo stesso mese; 1 se a e b hanno mesi diversi.
- c) -1 se il mese di a è tra gennaio e giugno e quello di b tra luglio e dicembre; 1 se il mese di b è tra gennaio e giugno e quello di a tra luglio e dicembre.
- d) -1 se il giorno oppure il mese di a è uguale a quello di b ; 1 se sia il giorno sia il mese di a sono diversi da quelli di b .

495. (Product, 2018-5-2)

[CROWDGRADER] Realizzare la classe Product, che rappresenta un prodotto di un supermercato, caratterizzato da descrizione e prezzo. I prodotti sono dotati di ordinamento naturale, in base alla loro descrizione (ordine alfabetico). Il metodo getMostExpensive restituisce il prodotto più costoso. Il campo comparatorByPrice contiene un comparatore tra prodotti, che confronta i prezzi.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Product a = new Product("Sale", 0.60),  
        b = new Product("Zucchero", 0.95),  
        c = new Product("Caffe'", 2.54);
```

```
System.out.println(Product.getMostExpensive());  
System.out.println(b.compareTo(c));  
System.out.println(Product.comparatorByPrice.compare(b, c));
```

Output:
Caffe', 2.54
1
-1

496. (Sphere Comparator, 2017-6-21)

La classe `Sphere` rappresenta una sfera nello spazio, caratterizzata dalle coordinate del centro e dal raggio. Due sfere sono uguali secondo `equals` se hanno lo stesso stesso raggio (indipendentemente dal centro). Dire quali delle seguenti sono specifiche valide per un comparatore tra due oggetti `Sphere a e b`. Dire anche quali specifiche sono coerenti con `equals`.

`compare(a,b)` restituisce (nei casi non contemplati, restituisce 0):

- a) -1 se il volume di a è minore di quello di b ; 1 se il volume di a è maggiore di quello di b .
- b) -1 se a e b hanno lo stesso centro; 1 se a e b hanno centri diversi.
- c) -1 se la somma dei due raggi è minore della distanza tra i centri; 1 se la somma dei due raggi è maggiore della distanza tra i centri.
- d) -1 se i raggi sono diversi; 1 se i raggi sono uguali, ma i centri sono diversi.

497. (**Book, 2016-7-21**)

Implementare la classe Book, che rappresenta un libro diviso in capitoli. Il metodo addChapter aggiunge un capitolo in coda al libro, caratterizzato da titolo e contenuto. I capitoli sono automaticamente numerati a partire da 1. Il metodo getChapterName(i) restituisce il titolo del capitolo *i*-esimo, mentre il metodo getChapterContent(i) ne restituisce il contenuto.

Gli oggetti Book devono essere clonabili. Inoltre, la classe deve essere dotata di ordinamento naturale, basato sul numero di capitoli.

L'implementazione deve rispettare il seguente esempio d'uso.

Esempio d'uso:

```
Book b = new Book();
b.addChapter("Prefazione", "Sono_passati_pochi_anni...");
b.addChapter("Introduzione", "Un_calcolatore_digitale...")
;
b.addChapter("Sistemi_di_elaborazione", "Un_calcolatore...
");
Book bb = b.clone();
System.out.println(bb.getChapterContent(1));
System.out.println(bb.getChapterTitle(2));
```

Output:

Sono passati pochi anni...
Introduzione

498. (**Set of Integer comparator, 2016-6-22**)

Dire quali delle seguenti sono specifiche valide per un Comparator tra due oggetti di tipo Set<Integer>, motivando la risposta. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

compare(a, b) restituisce:

- a) -1 se *a* contiene un numero minore di tutti i numeri di *b*; 1 se *b* contiene un numero minore di tutti i numeri di *a*
- b) -1 se la somma dei numeri di *a* è minore della somma dei numeri di *b*; 1 se la media dei numeri di *b* è maggiore della media dei numeri di *a*
- c) -1 se *a* contiene tutti numeri negativi e *b* no; 1 se *b* contiene tutti numeri positivi e *a* no
- d) -1 se *a* contiene il numero 0; 1 se *a* non contiene il numero 0
- e) -1 se *a* contiene il numero 0 e *b* no; 1 se *b* contiene il numero 0 e *a* no

499. (**Engine Comparator, 2016-4-21**)

Con riferimento alla classe dell'esercizio 2, dire quali delle seguenti sono specifiche valide per un Comparator tra due oggetti di tipo Engine, motivando la risposta. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

compare(a, b) restituisce:

- a) -1 se *a* ha cilindrata minore di *b*; 1 se *a* ha cilindrata maggiore di *b*
- b) -1 se *a* ha potenza minore della metà di *b*; 1 se *a* ha potenza maggiore del doppio di *b*
- c) -1 se *a* ha il rapporto potenza/cilindrata minore di *b*; 1 se *a* ha il rapporto potenza/cilindrata maggiore di *b*
- d) -1 se *a* ha cilindrata oppure potenza minore di *b*; 1 se *a* ha sia cilindrata sia potenza maggiori di *b*
- e) -1 se *a* ha cilindrata 1200 e potenza minore di *b*; 1 se *a* ha cilindrata 1200 e potenza maggiore di *b*

500. (**SetComparator, 2015-7-8**)

Valutare le seguenti specifiche per un comparatore tra insiemi, indicando quali sono valide e perché.

Dati due Set<?> *a* e *b*, compare(*a*,*b*) restituisce (nei casi non previsti, restituisce zero):

- a) -1 se *a* ∩ *b* = ∅ (*a* e *b* disgiunti); 1 se *a* ∩ *b* ≠ ∅.

- b) -1 se $a \subset b$ (a contenuto strettamente in b); 1 se $b \subset a$.
- c) -1 se a è vuoto e b no; 1 se a contiene un oggetto che non appartiene a b .
- d) -1 se a è un HashSet e b è un TreeSet; 1 se a è un TreeSet e b è un HashSet.

501. (Box, 2015-2-5)

Realizzare la classe `Box`, che rappresenta una scatola, caratterizzata dalle sue tre dimensioni: altezza, larghezza e profondità. Due scatole sono considerate uguali (da `equals`) se hanno le stesse dimensioni. Le scatole sono dotate di ordinamento naturale basato sul loro volume. Infine, il metodo `fitsIn`, invocato su una scatola x , accetta un'altra scatola y e restituisce `true` se e solo se y è sufficientemente grande da contenere x .

Esempio d'uso:

```
Box grande = new Box(20, 30, 40), grande2 = new Box(30, 20, 40),
    piccolo = new Box(10, 10, 50);
System.out.println(grande.equals(grande2));
System.out.println(grande.compareTo(piccolo));
System.out.println(piccolo . fitsIn (grande));
```

Output:
false
1
false

502. (DataSeries, 2015-1-20)

Realizzare la classe `DataSeries`, che rappresenta una serie storica di dati numerici (ad es., la popolazione di una regione anno per anno). Il metodo `set` imposta il valore della serie per un dato anno. Il metodo statico `comparatorByYear` accetta un anno e restituisce un comparatore tra serie di dati che confronta il valore delle due serie per quell'anno.

Esempio d'uso:

```
DataSeries pop1 = new DataSeries(),
    pop2 = new DataSeries();
pop1.set(2000, 15000.0); pop1.set(2005, 18500.0); pop1.set(2010, 19000.0);
pop2.set(2000, 12000.0); pop2.set(2005, 16000.0); pop2.set(2010, 21000.0);
Comparator<DataSeries> c2005 = DataSeries.comparatorByYear(2005),
    c2010 = DataSeries.comparatorByYear(2010);
System.out.println(c2005.compare(pop1, pop2));
System.out.println(c2010.compare(pop1, pop2));
```

Output:
1
-1

503. (EmployeeComparator, 2014-9-18)

Un employee è caratterizzato da nome (stringa) e salario (intero non negativo). Dire quali delle seguenti sono specifiche valide per un `Comparator` tra employee. In caso negativo, motivare la risposta con un controsenso. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

`compare(x,y)` restituisce:

- a) -1 se il nome di x è uguale a quello di y , ma i salari sono diversi; 1 se il salario di x è uguale a quello di y , ma i nomi sono diversi.
- b) -1 se il salario di x è zero e quello di y è maggiore di zero; 1 se il salario di y è zero e quello di x è maggiore di zero.
- c) -1 se il salario di x è *maggior*e di quello di y ; 1 se il salario di x è *minore* di quello di y .
- d) -1 se il nome di x precede alfabeticamente quello di y ed il salario di x è minore di quello di y ; 1 se il nome di y precede alfabeticamente quello di x .
- e) -1 se il salario di x è minore di 2000 e quello di y è maggiore di 1000; 1 se il salario di y è minore di 2000 e quello di x è maggiore di 1000.

504. (Playlist, 2014-7-28)

Implementare le classi `Song` e `Playlist`. Una canzone è caratterizzata dal nome e dalla durata in secondi. Una playlist è una lista di canzoni, compresi eventuali duplicati, ed offre il metodo `add`, che aggiunge una canzone in coda alla lista, e `remove`, che rimuove *tutte* le occorrenze di una

canzone dalla lista. Infine, la classe `Playlist` è dotata di ordinamento naturale basato sulla durata totale di ciascuna playlist.

Sono preferibili le implementazioni in cui il confronto tra due playlist avvenga in tempo costante.

| | |
|---|-----------------|
| Esempio d'uso: | Output: |
| <pre>Song one = new Song("One", 275), two = new Song("Two", 362); Playlist a = new Playlist(), b = new Playlist(); a.add(one); a.add(two); a.add(one); b.add(one); b.add(two); System.out.println(a.compareTo(b)); a.remove(one); System.out.println(a.compareTo(b));</pre> | <pre>1 -1</pre> |

505. (Pizza, 2014-11-3)

[CROWDGRADER] Realizzare la classe `Pizza`, in modo che ad ogni oggetto si possano assegnare degli ingredienti, scelti da un elenco fissato. Ad ogni ingrediente è associato il numero di calorie che apporta alla pizza. Gli oggetti di tipo `Pizza` sono dotati di ordinamento naturale, sulla base del numero totale di calorie. Infine, gli oggetti di tipo `Pizza` sono anche clonabili.

| | |
|--|--------------|
| Esempio d'uso: | Output: |
| <pre>Pizza margherita = new Pizza(), marinara = new Pizza(); margherita.addIngrediente(Pizza.Ingrediente.POMODORO); margherita.addIngrediente(Pizza.Ingrediente.MOZZARELLA); marinara.addIngrediente(Pizza.Ingrediente.POMODORO); marinara.addIngrediente(Pizza.Ingrediente.AGLIO); Pizza altra = margherita.clone(); System.out.println(altra.compareTo(marinara));</pre> | <pre>1</pre> |

506. (String comparator, 2013-6-25)

Dire quali delle seguenti sono specifiche valide per un `Comparator` tra due oggetti di tipo `String`, motivando la risposta. Nei casi non previsti dalle specifiche, il comparatore restituisce 0.

`compare(a, b)` restituisce:

- a) -1 se *a* contiene caratteri non alfabetici (ad es., numeri) e *b* no; 1 se *b* contiene caratteri non alfabetici (ad es., numeri) ed *a* no.
- b) -1 se *a* è lunga esattamente la metà di *b*; 1 se *a* è lunga esattamente il doppio di *b*.
- c) -1 se *a* è un prefisso proprio di *b* (cioè un prefisso diverso da *b*); 1 se *b* è un prefisso proprio di *a*.
- d) -1 se *b* comincia per maiuscola e *a* no; 1 se sia *a* sia *b* cominciano per maiuscola.
- e) -1 se *a* contiene la lettera "x" e *b* contiene la lettera "y"; 1 se *a* non contiene la lettera "x" e *b* non contiene la lettera "y".

507. (MaxBox, 2013-1-22)

Implementare la classe parametrica `MaxBox`, che funziona come un contenitore che conserva solo l'elemento "massimo" che vi sia mai stato inserito. L'ordinamento tra gli elementi può essere stabilito in due modi diversi: se al costruttore di `MaxBox` viene passato un oggetto `Comparator`, quell'oggetto verrà usato per stabilire l'ordinamento; altrimenti, verrà utilizzato l'ordinamento naturale fornito da `Comparable`. In quest'ultimo caso, se la classe degli elementi non implementa `Comparable`, viene sollevata un'eccezione.

| | |
|--|-----------------|
| Esempio d'uso: | Output: |
| <pre>MaxBox<String> bb1 = new MaxBox<String>(); MaxBox<String> bb2 = new MaxBox<String>(new Comparator<String>() { public int compare(String a, String b) { return a.length() - b.length(); } }); bb1.insert("dodici"); bb1.insert("sette"); bb2.insert("dodici"); bb2.insert("sette"); System.out.println(bb1.getMax()); System.out.println(bb2.getMax());</pre> | sette dodici |

508. (Point, 2012-6-18)

La classe Point rappresenta un punto del piano cartesiano con coordinate intere:

```
public class Point {
    private int x, y;
    ...
}
```

Spiegare quali delle seguenti sono implementazioni valide per il metodo `compare(Point a, Point b)` tratto dall'interfaccia `Comparator<Point>`, supponendo che tale metodo abbia accesso ai campi privati di Point.

- a) `return a.x-b.x;`
- b) `return a.x-b.y;`
- c) `return ((a.x*a.x)+(a.y*a.y)) - ((b.x*b.x)+(b.y*b.y));`
- d) `return (a.x-b.x)+(a.y-b.y);`
- e) `return (a.x-b.x)*(a.x-b.x)+ (a.y-b.y)*(a.y-b.y);`

509. (Time, 2010-9-14)

Implementare la classe Time, che rappresenta un orario della giornata (dalle 00:00:00 alle 23:59:59). Gli orari devono essere confrontabili secondo Comparable. Il metodo `minus` accetta un altro orario `x` come argomento e restituisce la differenza tra questo orario e `x`, sotto forma di un nuovo oggetto Time. La classe fornisce anche gli orari predefiniti MIDDAY e MIDNIGHT.

| | |
|--|----------------------------|
| Esempio d'uso: | Output dell'esempio d'uso: |
| <pre>Time t1 = new Time(14,35,0); Time t2 = new Time(7,10,30); Time t3 = t1.minus(t2); System.out.println(t3); System.out.println(t3.compareTo(t2)); System.out.println(t3.compareTo(Time.MIDDAY));</pre> | 7:24:30 1 -1 |

510. (PartiallyComparable, 2010-6-28)

L'interfaccia PartComparable (per *partially comparable*) rappresenta un tipo i cui elementi sono *parzialmente* ordinati.

```
public interface PartComparable<T> {
    public PartComparison compareTo(T x);
}

public enum PartComparison {
    SMALLER, EQUAL, GREATER, UNCOMP;
}
```

Implementare la classe `POSet` (per *partially ordered set*), che rappresenta un insieme parzialmente ordinato, i cui elementi implementano l'interfaccia `PartComparable`. Un oggetto di questo insieme è detto *massimale* se nessun altro oggetto nell'insieme è maggiore di lui.

Il metodo `add` aggiunge un oggetto all'insieme, mentre il metodo `isMaximal` restituisce vero se l'oggetto passato come argomento è uno degli oggetti massimali dell'insieme, restituisce falso se tale oggetto appartiene all'insieme, ma non è massimale, ed infine solleva un'eccezione se l'oggetto non appartiene all'insieme. Il metodo `isMaximal` deve terminare in tempo costante.

| | |
|---|---|
| <pre>// Stringhe, ordinate parzialmente dalla relazione di prefisso class POString implements PartComparable<POString> { ... } POSet<POString> set = new POSet<POString>(); set.add(new POString("architetto")); set.add(new POString("archimede")); set.add(new POString("archi")); set.add(new POString("bar")); set.add(new POString("bari")); System.out.println(set.isMaximal(new POString("archimede"))) ; System.out.println(set.isMaximal(new POString("bar"))); set.add(new POString("archimedeo")); System.out.println(set.isMaximal(new POString("archimede"))) ;</pre> | Output dell'esempio d'uso: true false false |
|---|---|

511. (Rebus, 2010-5-3)

In enigmistica, un rebus è un disegno dal quale bisogna ricostruire una frase. La traccia del rebus comprende anche la lunghezza di ciascuna parola della soluzione. Dato il seguente scheletro per la classe `Rebus`, dove `picture_name` è il nome del file che contiene il disegno (sempre diverso da `null`) e `word_length` è una lista di interi, che rappresentano la lunghezza di ciascuna parola nella soluzione,

```
class Rebus {
    private String picture_name;
    private List word_length;
}
```

considerare le seguenti specifiche alternative per un `Comparator` tra oggetti `Rebus`. Dati due `Rebus` `a` e `b`, `compare` deve restituire:

- a)
 - -1, se `a` e `b` hanno immagini diverse e la lunghezza totale della soluzione di `a` è minore di quella di `b`;
 - 0, se `a` e `b` hanno la stessa immagine;
 - 1, altrimenti.
- b)
 - -1, se `a` e `b` hanno immagini diverse e la soluzione di `b` contiene una parola più lunga di tutte le parole di `a`;
 - 0, se `a` e `b` hanno la stessa immagine, oppure le parole più lunghe delle due soluzioni hanno la stessa lunghezza;
 - 1, altrimenti.
- c)
 - -1, se nessuna delle due immagini ha estensione `.png`, e la soluzione di `a` contiene meno parole di quella di `b`;
 - 0, se almeno una delle due immagini ha estensione `.png`;
 - 1, altrimenti.

Dire se ciascun criterio di ordinamento è valido, giustificando la risposta. (15 punti)
Implementare il criterio (b). (15 punti)

512. (Version, 2010-2-24)

La classe Version rappresenta una versione di un programma. Una versione può avere due o tre parti intere ed, optionalmente, un'etichetta "alpha" o "beta". (15 punti)

La classe Version deve implementare l'interfaccia Comparable<Version>, in modo che una versione sia minore di un'altra se la sua numerazione è precedente a quella dell'altra. Le etichette "alpha" e "beta" non influiscono sull'ordinamento. (12 punti)

Rispettare il seguente caso d'uso, compreso il formato dell'output.

| | |
|---|--|
| <p>Esempio d'uso:</p> <pre>Version v1 = new Version(1, 0); Version v2 = new Version(2, 4, Version.alpha); Version v3 = new Version(2, 6, 33); Version v4 = new Version(2, 6, 34, Version.beta); System.out.println(v1); System.out.println(v2); System.out.println(v3); System.out.println(v4); System.out.println(v1.compareTo(v2)); System.out.println(v4.compareTo(v3));</pre> | <p>Output dell'esempio d'uso:</p> <pre>1.0 2.4alpha 2.6.33 2.6.34beta -1 1</pre> |
|---|--|

513. (2010-11-30)

Con riferimento all'esercizio 1, determinare quali tra i seguenti criteri sono validi per un Comparator tra segmenti, e perché. (18 punti)

Inoltre, implementare uno dei tre criteri. (8 punti)

Dati due Segment a e b, compare(a,b) deve restituire:

- a) • -1, se a è più corto di b;
 • 1, se b è più corto di a;
 • 0, altrimenti.
- b) • -1, se a, considerato come insieme di punti, è contenuto in b;
 • 1, se a non è contenuto in b, ma b è contenuto in a;
 • 0, altrimenti
- c) • -1, se a è orizzontale e b è verticale;
 • 1, se b è orizzontale e a è verticale;
 • 0, altrimenti.

514. (IncreasingSubsequence, 2009-9-l'8)

Implementare la classe IncreasingSubseq che, data una lista di oggetti tra loro confrontabili, rappresenta la *sottosequenza crescente* che inizia col primo elemento.

Attenzione: la classe deve funzionare con qualunque tipo di dato che sia confrontabile (non solo con "Integer").

Sarà valutato negativamente l'uso di "strutture di appoggio", ovvero di spazio aggiuntivo di dimensione non costante.

| | |
|--|---|
| <p>Esempio d'uso:</p> <pre>List<Integer> l = new LinkedList<Integer>(); l.add(10); l.add(3); l.add(5); l.add(12); l.add(11); l.add(35); for (Integer i: new IncreasingSubseq<Integer>(l)) System.out.println(i);</pre> | <p>Output dell'esempio d'uso:</p> <pre>10 12 35</pre> |
|--|---|

515. (Circle, 2009-4-23)

Nell'ambito di un programma di geometria, la classe `Circle` rappresenta una circonferenza sul piano cartesiano. Il suo costruttore accetta le coordinate del centro ed il valore del raggio. Il metodo `overlaps` prende come argomento un'altra circonferenza e restituisce vero se e solo se le due circonferenze hanno almeno un punto in comune.

Fare in modo che `Circle` implementi `Comparable`, con il seguente criterio di ordinamento: una circonferenza è “minore” di un’altra se è interamente contenuta in essa, mentre se nessuna delle due circonferenze è contenuta nell’altra, esse sono considerate “uguali”. Dire se tale criterio di ordinamento è valido, giustificando la risposta.

| Esempio d’uso: | Output dell’esempio d’uso: |
|--|----------------------------|
| <pre>Circle c1 = new Circle(0,0,2); Circle c2 = new Circle(1,1,1); System.out.println(c1.overlaps(c2)); System.out.println(c1.compareTo(c2));</pre> | <pre>true 0</pre> |

516. (Triangle 2, 2009-11-27)

La classe `Triangle` rappresenta un triangolo. Il suo costruttore accetta la misura dei suoi lati, e lancia un’eccezione se tali misure non danno luogo ad un triangolo. Il metodo `equals` stabilisce se due triangoli sono isometrici (uguali). Il metodo `similar` stabilisce se due triangoli sono simili (hanno gli stessi angoli, ovvero lo stesso rapporto tra i lati).

Il metodo `perimeterComparator` restituisce un comparatore che confronta i triangoli in base al loro perimetro.

Nota: tre numeri positivi x , y e z possono essere le misure dei lati di un triangolo a patto che $x < y + z$, $y < x + z$ e $z < x + y$.

| Esempio d’uso: | Output dell’esempio d’uso: |
|--|------------------------------|
| <pre>Triangle a = new Triangle(3,4,5); Triangle b = new Triangle(4,5,3); Triangle c = new Triangle(8,6,10); System.out.println(a.equals(b)); System.out.println(a.similar(b)); System.out.println(a.similar(c)); Comparator<Triangle> pc = Triangle. perimeterComparator(); System.out.println(pc.compare(b, c));</pre> | <pre>true true true -1</pre> |

517. (2009-1-15)

La seguente classe A fa riferimento ad una classe B. Implementare la classe B in modo che venga compilata correttamente, permetta la compilazione della classe A e produca l’output indicato.

Inoltre, rispondere alle seguenti domande:

- Quale design pattern si ritrova nel metodo `Collections.sort`?
- Quale ordinamento sui numeri interi realizza la vostra classe B?

| | |
|---|---|
| <pre> public class A { public static void main(String[] args) { List<Integer> l = new LinkedList<Integer>(); l.add(3); l.add(70); l.add(23); l.add(50); l.add(5); l.add (20); Collections.sort(l, new B()); for (Integer i: l) System.out.println(i); } } </pre> | Output richiesto: 20 50 70 3 5 23 |
|---|---|

518. (Sorter, 2008-1-30)

Implementare una classe parametrica Sorter, con un solo metodo check. Il metodo check confronta l'oggetto che riceve come argomento con quello che ha ricevuto alla chiamata precedente, o con quello passato al costruttore se si tratta della prima chiamata a check. Il metodo restituisce -1 se il nuovo oggetto è più piccolo del precedente, 1 se il nuovo oggetto è più grande del precedente e 0 se i due oggetti sono uguali. Per effettuare i confronti, Sorter si basa sul fatto che il tipo usato come parametro implementi l'interfaccia Comparable.

| | |
|--|--|
| Esempio d'uso: <pre> Sorter<Integer> s = new Sorter<Integer>(7); System.out.println(s.check(4)); System.out.println(s.check(1)); System.out.println(s.check(6)); System.out.println(s.check(6)); </pre> | Output dell'esempio d'uso: -1 -1 1 0 |
|--|--|

519. (FunnyOrder, 2007-9-17)

Determinare l'output del seguente programma e descrivere brevemente l'ordinamento dei numeri interi definito dalla classe FunnyOrder.

```

public class FunnyOrder implements Comparable<FunnyOrder> {
    private int val;
    public FunnyOrder(int n) { val = n; }
    public int compareTo(FunnyOrder x) {
        if (val%2 == 0 && x.val%2 != 0) return -1;
        if (val%2 != 0 && x.val%2 == 0) return 1;
        if (val < x.val) return -1;
        if (val > x.val) return 1;
        return 0;
    }
    public static void main(String[] args) {
        List<FunnyOrder> l = new LinkedList<FunnyOrder>();
        l.add(new FunnyOrder(16));
        l.add(new FunnyOrder(3));
        l.add(new FunnyOrder(4));
        l.add(new FunnyOrder(10));
        l.add(new FunnyOrder(2));
        Collections.sort(l);
        for (FunnyOrder f: l)
            System.out.println(f.val + " ");
    }
}

```

520. (Rational, 2007-6-29)

- (18 punti) Si implementi una classe Rational che rappresenti un numero razionale in maniera esatta. Il costruttore accetta numeratore e denominatore. Se il denominatore è negativo, viene lanciata una eccezione. Il metodo plus prende un altro Rational x come argomento e restituisce la somma di this e x . Il metodo times prende un altro Rational x come argomento e restituisce il prodotto di this e x .
- (9 punti) La classe deve assicurarsi che numeratore e denominatore siano sempre ridotti ai minimi termini. (Suggerimento: la minimizzazione della frazione può essere compito del costruttore)
- (7 punti) La classe deve implementare l'interfaccia Comparable<Rational>, in base al normale ordinamento tra razionali.

| | |
|---|---|
| <p>Esempio d'uso:</p> <pre>Rational n = new Rational(2,12); // due dodicesimi Rational m = new Rational(4,15); // quattro quindicesimi Rational o = n.plus(m); Rational p = n.times(m); System.out.println(n); System.out.println(o); System.out.println(p);</pre> | <p>Output dell'esempio d'uso:</p> <p>1/6 13/30 2/45</p> |
|---|---|

19 Indice Cronologico

- **2006-4-27:** Average, pag. 68 ; Esercizio 195, pag. 52 ; BinaryTreePreIterator, pag. 198 ; Esercizio 197, pag. 105 ;
- **2006-6-26:** Moto bidimensionale, pag. 67 ; Esercizio 189, pag. 51 ; Publication, pag. 91 ; Esercizio 191, pag. 161 ; DoubleQueue, pag. 91 ; Esercizio 193, pag. 92 105 118 ;
- **2006-7-17:** Moto accelerato, pag. 67 ; Esercizio 165, pag. 51 ; Spartito, pag. 90 ; Esercizio 167, pag. 161 ; TwoSteps, pag. 197 ; Esercizio 169, pag. 90 104 117 ;
- **2006-9-15:** FallingBody, pag. 66 ; Esercizio 171, pag. 50 ; TreeType, pag. 66 166 ; Esercizio 173, pag. 160 ; SuperclassIterator, pag. 170 197 ; Esercizio 175, pag. 104 193 ;
- **2007-1-12:** Polinomio, pag. 66 ; Esercizio 185, pag. 49 ; Esercizio 186, pag. 160 ; Insieme di polinomi, pag. 90 ;
- **2007-2-23:** Inventory, pag. 89 117 ; Primes, pag. 197 ; Esercizio 182, pag. 49 ; Esercizio 183, pag. 159 ;
- **2007-2-7:** Polinomio bis, pag. 89 117 ; Monomio, pag. 7 ; Esercizio 178, pag. 48 ; Esercizio 179, pag. 159 ;
- **2007-4-26:** Genealogia, pag. 65 ; AncestorIterator, pag. 197 ; Esercizio 41, pag. 48 ; Esercizio 42, pag. 65 103 ;
- **2007-6-29:** Rational, pag. 64 208 ; Esercizio 54, pag. 47 ; Polinomio su un campo generico, pag. 116 ; Esercizio 56, pag. 88 103 ; Esercizio 57, pag. 159 ; Highway, pag. 89 193 ;
- **2007-7-20:** CommonDividers, pag. 115 196 ; Esercizio 34, pag. 47 ; ParkingLot, pag. 64 115 ; Esercizio 36, pag. 103 116 ; Esercizio 37, pag. 158 ; Simulazione di ParkingLot, pag. 193 ;
- **2007-9-17:** Aereo, pag. 64 ; Esercizio 44, pag. 46 ; Selector, pag. 114 196 ; FunnyOrder, pag. 88 115 208 ; Esercizio 47, pag. 158 ;
- **2008-1-30:** Recipe, pag. 87 ; Esercizio 65, pag. 46 ; Sorter, pag. 114 208 ; Esercizio 67, pag. 63 124 ; Esercizio 68, pag. 158 ;
- **2008-2-25:** BoolExpr, pag. 87 113 ; Esercizio 60, pag. 45 ; MyFor, pag. 114 196 ; Esercizio 62, pag. 63 123 ; Esercizio 63, pag. 157 ;
- **2008-3-27:** Impianto e Apparecchio, pag. 63 86 ; Esercizio 49, pag. 45 ; DelayIterator, pag. 193 ; Esercizio 51, pag. 123 ; Esercizio 52, pag. 157 ;
- **2008-4-21:** Triangolo, pag. 62 131 ; Esercizio 251, pag. 44 ; Esercizio 252, pag. 7 ; CrazyIterator, pag. 195 ;
- **2008-6-19:** Molecola, pag. 86 ; Esercizio 237, pag. 44 ; RunnableWithProgress, pag. 192 ; Esercizio 239, pag. 123 ; Esercizio 240, pag. 157 ;
- **2008-7-9:** Esercizio 241, pag. 7 ; Esercizio 242, pag. 43 ; MutexWithLog, pag. 192 ; Esercizio 244, pag. 122 ; Esercizio 245, pag. 156 ;
- **2008-9-8:** PostIt, pag. 86 ; Esercizio 260, pag. 43 ; RunnableWithArg, pag. 191 ; Esercizio 262, pag. 62 ; Esercizio 263, pag. 156 ;
- **2009-1-15:** Anagramma, pag. 62 ; Esercizio 232, pag. 42 ; Volo e Passeggero, pag. 85 ; Esercizio 234, pag. 122 207 ; Esercizio 235, pag. 155 ;

- **2009-1-29:** Interval, pag. 62 130 ; Esercizio 247, pag. 42 ; Split, pag. 113 ; Esercizio 249, pag. 155 ;
- **2009-11-27:** Triangle 2, pag. 207 ; Esercizio 204, pag. 41 ; CountByType, pag. 85 169 ; Esercizio 206, pag. 122 130 ; Esercizio 207, pag. 155 ;
- **2009-2-19:** Container, pag. 85 ; Esercizio 255, pag. 41 ; Interleave, pag. 113 ; Esercizio 257, pag. 122 ; Esercizio 258, pag. 154 ;
- **2009-4-23:** UML, pag. 85 ; Esercizio 223, pag. 40 ; Circle, pag. 61 207 ; Esercizio 225, pag. 121 130 ;
- **2009-6-19:** Tutor, pag. 84 ; Esercizio 199, pag. 40 ; Cardinal, pag. 135 ; Esercizio 201, pag. 113 121 ; Esercizio 202, pag. 154 ;
- **2009-7-9:** Washer, pag. 129 ; Esercizio 209, pag. 39 ; Elevator, pag. 191 ; Esercizio 211, pag. 113 121 ; Esercizio 212, pag. 154 ;
- **2009-9-1'8:** Auction, pag. 190 ; Esercizio 227, pag. 39 ; IncreasingSubsequence, pag. 195 206 ; Esercizio 229, pag. 120 129 ; Esercizio 230, pag. 153 ;
- **2010-1-22:** Color, pag. 61 84 ; Esercizio 214, pag. 38 ; GetByType, pag. 84 169 ; Esercizio 216, pag. 120 129 166 ; Esercizio 217, pag. 153 ;
- **2010-11-30:** Segment, pag. 61 166 ; Esercizio 29, pag. 38 ; Esercizio 30, pag. 206 ; SelectKeys, pag. 84 112 ; Esercizio 32, pag. 152 ;
- **2010-2-24:** Wall, pag. 60 ; Esercizio 219, pag. 37 ; Version, pag. 206 ; Esercizio 221, pag. 152 ;
- **2010-5-3:** Crosswords, pag. 60 ; Esercizio 7, pag. 37 ; Rebus, pag. 205 ;
- **2010-6-28:** QueueOfTasks, pag. 190 ; Esercizio 10, pag. 36 ; PartiallyComparable, pag. 83 204 ; Esercizio 12, pag. 152 ;
- **2010-7-26:** Tetris, pag. 59 ; Esercizio 2, pag. 36 ; TetrisPiece, pag. 135 ; Esercizio 4, pag. 112 120 ; Esercizio 5, pag. 151 ;
- **2010-9-14:** Time, pag. 59 204 ; Esercizio 24, pag. 35 ; Intersect, pag. 83 112 ; ExecuteInParallel, pag. 190 ; Esercizio 27, pag. 151 ;
- **2011-2-7:** VoteBox, pag. 190 ; Esercizio 14, pag. 35 ; Operai, pag. 6 ; MakeMap, pag. 83 112 ; Esercizio 17, pag. 151 ;
- **2011-3-4:** MultiProgressBar, pag. 189 ; Esercizio 19, pag. 34 ; Esercizio 20, pag. 112 120 ; PrintBytes, pag. 59 ; Esercizio 22, pag. 150 ;
- **2012-4-23:** Esercizio 264, pag. 34 ; Safe, pag. 58 ; Esercizio 266, pag. 6 ; Panino, pag. 82 135 ;
- **2012-6-18:** Esercizio 273, pag. 33 ; Point, pag. 204 ; BoundedMap, pag. 82 111 ; ThreadRace, pag. 189 ; Esercizio 277, pag. 150 ;
- **2012-7-9:** Mystery thread, pag. 189 ; Esercizio 269, pag. 32 ; Social network, pag. 82 ; NumberType, pag. 134 ; Esercizio 272, pag. 149 ;
- **2012-9-3:** Esercizio 287, pag. 32 ; Mystery thread 2, pag. 188 ; Anagrammi, pag. 6 165 ; Bijection, pag. 81 ; Esercizio 291, pag. 149 ;
- **2013-1-22:** Esercizio 292, pag. 32 ; Shared object, pag. 188 ; Insieme di lettere, pag. 6 165 ; MaxBox, pag. 203 ; Esercizio 296, pag. 149 ;
- **2013-12-16:** Esercizio 154, pag. 31 ; Note, pag. 134 ; concurrentMax, pag. 188 ; agree, pag. 101 ; Esercizio 158, pag. 149 ;
- **2013-2-11:** Esercizio 278, pag. 30 ; MultiSet, pag. 5 81 ; Concurrent filter, pag. 187 ; Esercizio 281, pag. 148 ;

- **2013-3-22:** Esercizio 282, pag. 30 ; Auditorium, pag. 81 ; Cane, pag. 5 ; Shared average, pag. 187 ; Esercizio 286, pag. 148 ;
- **2013-4-29:** Esercizio 130, pag. 29 ; City, pag. 80 ; Pair, pag. 111 ; Esercizio 133, pag. 5 ;
- **2013-6-25:** Esercizio 134, pag. 29 ; String comparator, pag. 203 ; MultiBuffer, pag. 80 186 ; Concat, pag. 80 101 ; Esercizio 138, pag. 148 ;
- **2013-7-9:** Esercizio 159, pag. 28 ; BloodType, pag. 134 ; processArray, pag. 186 ; isSorted, pag. 79 100 ; Esercizio 163, pag. 148 ;
- **2013-9-25:** Esercizio 139, pag. 28 ; Movie, pag. 79 ; executeWithDeadline, pag. 186 ; composeMaps, pag. 79 100 ; Esercizio 143, pag. 147 ;
- **2014-1-31:** Esercizio 149, pag. 27 ; BoundedSet, pag. 78 ; PostOfficeQueue, pag. 185 ; isMax, pag. 100 ; Esercizio 153, pag. 147 ;
- **2014-11-28:** Esercizio 82, pag. 27 ; Coin, pag. 134 ; Alarm, pag. 185 ; product, pag. 100 ; Esercizio 86, pag. 147 ;
- **2014-11-3:** Esercizio 102, pag. 26 ; Pizza, pag. 128 133 203 ; FunnyIterator, pag. 195 ;
- **2014-3-5:** Esercizio 144, pag. 26 ; PeriodicTask, pag. 4 185 ; Status, pag. 133 ; extractPos, pag. 99 ; Esercizio 148, pag. 146 ;
- **2014-4-28:** Esercizio 69, pag. 25 ; Shape, pag. 4 ; Shape equals, pag. 4 ;
- **2014-7-28:** Esercizio 77, pag. 25 ; Playlist, pag. 58 202 ; PriorityExecutor, pag. 184 ; inverseMap, pag. 78 99 ; Esercizio 81, pag. 146 ;
- **2014-7-3:** Esercizio 72, pag. 24 ; NutrInfo, pag. 133 ; Exchanger, pag. 184 ; subMap, pag. 78 99 ; Esercizio 76, pag. 146 ;
- **2014-9-18:** Esercizio 87, pag. 24 ; EmployeeComparator, pag. 202 ; Contest, pag. 77 ; atLeastOne, pag. 184 ; Esercizio 91, pag. 146 ;
- **2015-1-20:** Esercizio 92, pag. 23 ; DataSeries, pag. 202 ; Relation, pag. 77 111 ; difference, pag. 98 ; Esercizio 96, pag. 145 ;
- **2015-2-5:** Esercizio 110, pag. 23 ; Box, pag. 58 202 ; ForgetfulSet, pag. 183 ; reverseList, pag. 98 ; Esercizio 114, pag. 145 ;
- **2015-6-24:** Esercizio 105, pag. 22 ; SimpleThread, pag. 183 ; Controller, pag. 76 128 ; listIntersection, pag. 98 ; Esercizio 109, pag. 145 ;
- **2015-7-8:** Esercizio 97, pag. 22 ; TimeToFinish, pag. 183 ; Question e Answer, pag. 57 ; SetComparator, pag. 201 ; Esercizio 101, pag. 145 ;
- **2015-9-21:** Esercizio 115, pag. 21 ; Progression, pag. 76 ; StringQuiz, pag. 182 ; splitList, pag. 98 ; Esercizio 119, pag. 144 ;
- **2016-1-27:** Esercizio 120, pag. 21 ; Curriculum, pag. 75 127 ; twoPhases, pag. 182 ; Esercizio 123, pag. 110 119 ; Esercizio 124, pag. 144 ;
- **2016-3-3:** Esercizio 125, pag. 20 ; GameLevel, pag. 57 75 ; MysteryThread3, pag. 182 ; Soldier, pag. 3 ; Esercizio 129, pag. 144 ;
- **2016-4-21:** Esercizio 297, pag. 20 ; Engine, pag. 3 127 ; Engine Comparator, pag. 201 ; Count, pag. 107 ;
- **2016-6-22:** Esercizio 301, pag. 19 ; Set of Integer comparator, pag. 201 ; BlockingArray, pag. 181 ; arePermutations, pag. 97 ; Esercizio 305, pag. 144 ;
- **2016-7-21:** Esercizio 306, pag. 19 ; Book, pag. 57 165 201 ; findString, pag. 181 ; Esercizio 309, pag. 110 119 ; Esercizio 310, pag. 143 ;

- **2016-9-20:** Esercizio 311, pag. 18 ; SocialUser, pag. 75 ; Somma due, pag. 181 ; Esercizio 314, pag. 143 ;
- **2017-1-25:** Esercizio 315, pag. 18 ; LengthUnit, pag. 133 ; mergeIfSorted, pag. 75 181 ; Esercizio 318, pag. 143 ;
- **2017-10-6:** Esercizio 341, pag. 17 ; Clinica, pag. 74 ; Somma e azzera, pag. 180 ; Esercizio 344, pag. 142 ;
- **2017-2-23:** Esercizio 319, pag. 17 ; Polygon, pag. 3 ; sumAndMax, pag. 180 ; Esercizio 322, pag. 142 ;
- **2017-3-23:** Esercizio 323, pag. 16 ; BinRel, pag. 74 ; Bonus per Employee, pag. 179 ; Esercizio 326, pag. 142 ;
- **2017-4-26:** Esercizio 327, pag. 16 ; Room, pag. 73 ; Room equals, pag. 2 ; Generic constructor, pag. 110 ;
- **2017-6-21:** Esercizio 331, pag. 15 ; Sphere Comparator, pag. 200 ; Market, pag. 179 ; findNext, pag. 97 ; Esercizio 335, pag. 141 ;
- **2017-7-20:** Esercizio 336, pag. 15 ; Cartella, pag. 56 ; MysteryThread4, pag. 178 ; common-Keys, pag. 97 ; Esercizio 340, pag. 141 ;
- **2018-1-24:** Bug, pag. 73 ; Shared total, pag. 178 ; isIncreasing, pag. 97 ; Esercizio 348, pag. 141 ;
- **2018-10-18:** Esercizio 374, pag. 14 ; Component e Configuration, pag. 72 ; greatestLowerBound, pag. 96 ; Esercizio 377, pag. 141 ;
- **2018-2-22:** Book e Library, pag. 72 ; Two threads, pag. 178 ; cartesianProduct, pag. 96 ; Esercizio 356, pag. 140 ;
- **2018-3-23:** Esercizio 349, pag. 14 ; Studente, pag. 2 127 ; isSetSmaller, pag. 71 ; Esercizio 352, pag. 140 ;
- **2018-5-2:** Esercizio 357, pag. 13 ; Product, pag. 199 ; Merge, pag. 71 107 ;
- **2018-6-20:** Esercizio 365, pag. 12 ; Date, pag. 199 ; PeriodicExecutor, pag. 177 ; makeMap, pag. 96 ; Esercizio 369, pag. 140 ;
- **2018-7-19:** Esercizio 360, pag. 12 ; Fraction, pag. 2 ; SafeSet, pag. 71 176 ; MysteryThread5, pag. 177 ; Esercizio 364, pag. 139 ;
- **2018-9-17:** Esercizio 370, pag. 12 ; Cellphone, pag. 70 ; SharedCounter, pag. 176 ; Esercizio 373, pag. 139 ;
- **2019-1-23:** Esercizio 378, pag. 11 ; GuessTheNumber, pag. 176 ; findPrevious, pag. 95 ; Esercizio 381, pag. 139 ;
- **2019-10-9:** Shop, pag. 176 ; Esercizio 409, pag. 119 163 ; interleave2, pag. 95 ; Esercizio 411, pag. 139 ;
- **2019-2-15:** Esercizio 382, pag. 11 ; Range, pag. 109 ; Missing synch 2, pag. 175 ; Esercizio 385, pag. 138 ;
- **2019-3-19:** Esercizio 386, pag. 10 ; Library, pag. 70 ; Missing synch 3, pag. 175 ; Esercizio 389, pag. 138 ;
- **2019-4-29:** Esercizio 390, pag. 10 ; RotatingList, pag. 69 ; Rotating list comparator, pag. 199 ; Lambda, pag. 163 ;
- **2019-6-24:** Esercizio 394, pag. 9 ; Box e ColoredBox, pag. 1 ; SortedList, pag. 69 ; keysWithValue, pag. 95 ; Esercizio 398, pag. 138 ;

- **2019-7-23:** Esercizio 399, pag. 9 ; Student, pag. 1 ; RandomExecutor, pag. 174 ; Minimum enum, pag. 95 ; Esercizio 403, pag. 138 ;
- **2019-9-20:** Microwave, pag. 126 ; disjoin, pag. 94 ; MysteryThread6, pag. 173 ; Esercizio 407, pag. 137 ;
- **2020-1-24:** Product e Cart, pag. 69 ; MysteryThread7, pag. 173 ; Uguaglianza tra cart, pag. 1 ; Esercizio 415, pag. 137 ;
- **2020-2-27:** Accumulator, pag. 109 ; MysteryThread8, pag. 173 ; keysWithHighestValue, pag. 94 ; Esercizio 419, pag. 137 ;
- **2021-10-26:** WiFi, pag. 55 ; Missing synch 4, pag. 172 ; Esercizio 430, pag. 169 ; countInBetween, pag. 94 ;
- **2021-7-26:** GreenPass, pag. 55 ; InternalLayout1, pag. 126 ; overridingMap, pag. 94 ; Esercizio 423, pag. 171 ;
- **2021-9-24:** Radio, pag. 54 ; InternalLayout2, pag. 125 ; MysteryThread9, pag. 171 ; countOccurrences, pag. 93 ;
- **2022-1-26:** Exchange, pag. 54 ; Esercizio 433, pag. 1 ; Esercizio 434, pag. 169 ; combine, pag. 93 ;
- **2022-2-24:** FilteredSet, pag. 53 ; InternalLayout3, pag. 125 ; keysWithMaxValue, pag. 93 ;
- **2022-3-28:** Angle, pag. 53 ; InternalLayout4, pag. 125 ; parallelMax, pag. 171 ; Esercizio 442, pag. 137 ;

Java: Classi interne

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli "Federico II"

Corso di Linguaggi di Programmazione I

- Java permette di definire una classe (o interfaccia) all'interno di un'altra
- Queste classi vengono chiamate “interne” o “annidate”
 - in inglese, il termine usato è *nested*
 - In particolare, le classi interne non statiche sono chiamate *inner*
- Tale meccanismo arricchisce le possibilità di relazioni tra classi, introducendo in particolare **nuove regole di visibilità**

Le classi interne (non statiche) godono delle seguenti proprietà distinctive:

1) Privilegi di visibilità rispetto alla classe contenitrice e alle altre classi in essa contenute

- permettono una stretta collaborazione tra queste classi

2) Restrizioni di visibilità rispetto alle classi esterne a quella contenitrice

- permettono di nascondere la classe all'esterno (incapsulamento)

3) Un **riferimento隐式** ad un oggetto della classe contenitrice

- ogni oggetto della classe interna "conosce" l'oggetto della classe contenitrice che l'ha creato

- Oltre a campi e metodi, una classe può contenere altre classi o interfacce, dette "interne"
- Una classe che non sia interna viene chiamata "top-level"

- A differenza delle classi top-level, le classi interne possono avere **tutte le quattro visibilità** ammesse dal linguaggio
- La visibilità di una classe interna X stabilisce quali classi possono utilizzarla (cioè, istanziarla, estenderla, dichiarare riferimenti o parametri di tipo X, etc.)

Consideriamo il seguente esempio:

```
public class A {  
    private class B {  
        ...  
    }  
    class C {  
        ...  
    }  
}
```

La classe B **non è visibile al di fuori di A**

La classe C è visibile a tutte le classi che si trovano nello stesso pacchetto di A

```
public class A {  
    private class B {  
        ...  
    }  
    class C {  
        ...  
    }  
}
```

- Dall'esterno di A, i nomi completi delle classi B e C sono A.B e A.C, rispettivamente
- La visibilità di una classe interna non ha alcun effetto sul codice che si trova all'interno della classe che la contiene
- Ad esempio, la classe B dell'esempio è visibile a tutto il codice contenuto in A, compreso il codice contenuto in altre classi interne ad A, come ad esempio C
- Lo stesso discorso si applica per i campi e i metodi di una classe interna
 - i loro attributi di visibilità hanno effetto solo sul codice *esterno* alla classe contenitrice
- In altre parole, **tra classi contenute nella stessa classe non vige alcuna restrizione di visibilità**

- Ciascun oggetto di una classe interna (non statica, come spiegato dopo) possiede un **riferimento implicito** ad un oggetto della classe contenitrice
 - Tale riferimento viene inizializzato automaticamente al momento della creazione dell'oggetto
 - Tale riferimento non può essere modificato
-
- Supponiamo che B sia una classe interna di A
 - All'interno della classe B, la sintassi per denotare questo riferimento隐式 è **A.this**
 - L'uso di A.this è facoltativo, come quello di this
 - cioè, B può accedere ad un campo o metodo "f" della classe A sia con "A.f" che con "f"

Esempio:

```
public class A {  
    private int x;  
  
    public class B {  
        private int y;  
        public void stampami() {  
            System.out.println(A.this.x); // uso del riferimento implicito  
            System.out.println(y);  
        }  
    }  
}
```

- Il **contesto statico** di una classe è quella porzione del codice di quella classe che si trova nell'ambito di validità di un modificatore static
- Quindi, in una delle seguenti posizioni:
 - All'interno di un metodo statico
 - All'interno di un blocco di inizializzazione statico
 - Nella definizione di un attributo statico
- La parte restante della classe si chiama **contesto non statico**

- Supponiamo che B sia una classe interna di A
- Se viene creato un oggetto di tipo B in un *contesto non statico* della classe A, il riferimento implicito verrà inizializzato con il valore corrente di this
- In tutti gli altri casi, è necessario utilizzare una **nuova forma dell'operatore “new”**, ovvero:

<riferimento ad oggetto di tipo A>.new B(...)

Esempio: creazione di un oggetto di classe interna in un *contesto non statico* della classe esterna

```
public class A {  
    private int x;  
  
    public class B {  
        private int y;  
        public void stampami() {  
            System.out.println(A.this.x); // uso del riferimento implicito  
            System.out.println(y);  
        }  
    }  
  
    public B makeB(int val) {  
        B b = new B(); // il nuovo oggetto B è legato a this  
        b.y = val; // privilegi di visibilità  
        return b;  
    }  
}
```

Esempio: creazione di un oggetto di classe interna in un contesto *statico* della classe esterna

```
public class A {  
    private int x;  
  
    public class B {  
        private int y;  
        public void stampami() {  
            System.out.println(A.this.x); // uso del riferimento implicito  
            System.out.println(y);  
        }  
    }  
  
    public static void main(String[] args) {  
        B b = new B(); // errore di compilazione  
        A a = new A();  
        B b = a.new B(); // OK  
    }  
}
```

Classi interne statiche

- Le classi interne possono essere statiche o meno
- Una classe interna dichiarata nello scope di classe (cioè al di fuori di metodi e inizializzatori) è statica se preceduta dal modificatore “static”
- Le classi interne possono anche trovarsi all'interno di un metodo (parleremo di classi *locali*), ma non ne parleremo in questo corso
- **Le classi interne statiche non possiedono il riferimento implicito alla classe contenitrice**
- Nota: prima di Java 16, una classe interna non statica non poteva avere membri (campi o metodi) statici

- Riassumendo, le classi interne non statiche godono delle seguenti proprietà distintive:

1) Privilegi di visibilità rispetto alla classe contenitrice e alle altre classi in essa contenute

- permettono una stretta collaborazione tra queste classi

2) Restrizioni di visibilità rispetto alle classi esterne a quella contenitrice

- permettono di nascondere la classe all'esterno (incapsulamento)

3) Un riferimento implicito ad un oggetto della classe contenitrice

- ogni oggetto della classe interna “conosce” l’oggetto della classe contenitrice che l’ha creato

- Le classi interne *statiche* godono solo delle **prime due** proprietà

Nota: alcuni testi si riferiscono a tutte le classi interne come “annidate” e riservano il termine “interne” solo alle classi interne non statiche

Il seguente esempio mostra come le classi interne (anche statiche) abbiano pieno accesso ai membri privati della classe contenitrice

```
public class External
{
    private int n = 42;

    public static class A {
        public void foo(External guest) {
            // This is legal
            guest.n = 0; ←
        }
    }
    public static class B extends External {
        public void foo() {
            // Wrong syntax (compile-time error)
            n = 0;
            // Wrong syntax (compile-time error)
            this.n = 0;
            // This is legal
            super.n = 0; ←
        }
    }
}
```

Se le classi A e B fossero top-level, non potrebbero accedere al campo *n*.

- La figura rappresenta il memory layout delle classi definite di seguito
- Notate in particolare la differenza tra classe interna e sottoclasse

```
public class A {  
    private int x;  
    private class B {  
        private int y;  
    }  
    static class C {  
        private double z;  
    }  
    public static class D extends A {  
        private int w;  
    }  
    public static void main(String[] args) {  
        A a = new A();  
        B b = a.new B();  
        C c = new C();  
        A d = new D();  
    }  
}
```

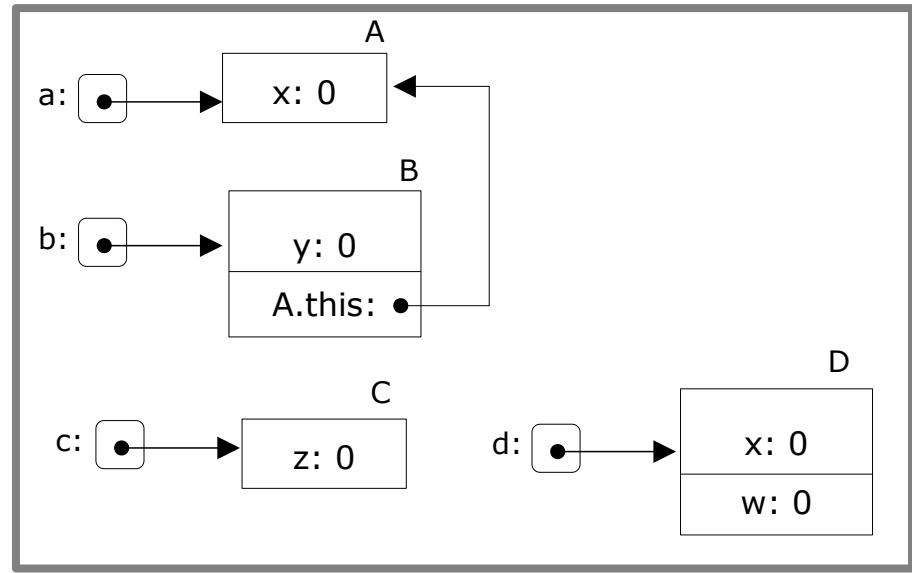


Figura 1: Il memory layout del frammento di programma a sinistra.

- C++

- [Come Java] Prevede classi interne (nested) con restrizioni di visibilità arbitraria (ad es., private)
- [Come Java] La classe interna ha accesso a tutti i membri della contenitrice
- [Diverso] La classe esterna *non* ha accesso privilegiato al contenuto di quella interna
- [Diverso] Non prevede riferimento implicito alla classe esterna
- Privilegi di visibilità si ottengono anche tramite il costrutto *friend*

- C#

- Regole analoghe a C++
- Nessun costrutto friend
- Le linee guida MS sconsigliano esplicitamente di creare classi interne pubbliche

Nell'ambito di un programma di geometria, si implementi la classe Triangolo, il cui costruttore accetta le misure dei tre lati. Se tali misure non danno luogo ad un triangolo, il costruttore deve lanciare un'eccezione. Il metodo getArea restituisce l'area di questo triangolo.

Si implementino anche la classe Triangolo.Rettangolo, il cui costruttore accetta le misure dei due cateti, e la classe Triangolo.Iisoscele, il cui costruttore accetta le misure della base e di uno degli altri lati.

Si ricordi che:

- Tre numeri a, b e c possono essere i lati di un triangolo a patto che $a < b+c$, $b < a+c$ e $c < a+b$.
- L'area di un triangolo di lati a, b e c è data da: $\sqrt{p(p-a)(p-b)(p-c)}$ (formula di Erone), dove p è il semiperimetro.

Output dell'esempio d'uso:

Esempio d'uso (fuori dalla classe Triangolo):

94.9918

19.9999

12.0

```
Triangolo x = new Triangolo(10,20,25);
Triangolo y = new Triangolo.Rettangolo(5,8);
Triangolo z = new Triangolo.Iisoscele(6,5);
```

```
System.out.println(x.getArea());
System.out.println(y.getArea());
System.out.println(z.getArea());
```

Polimorfismo

Polimorfismo

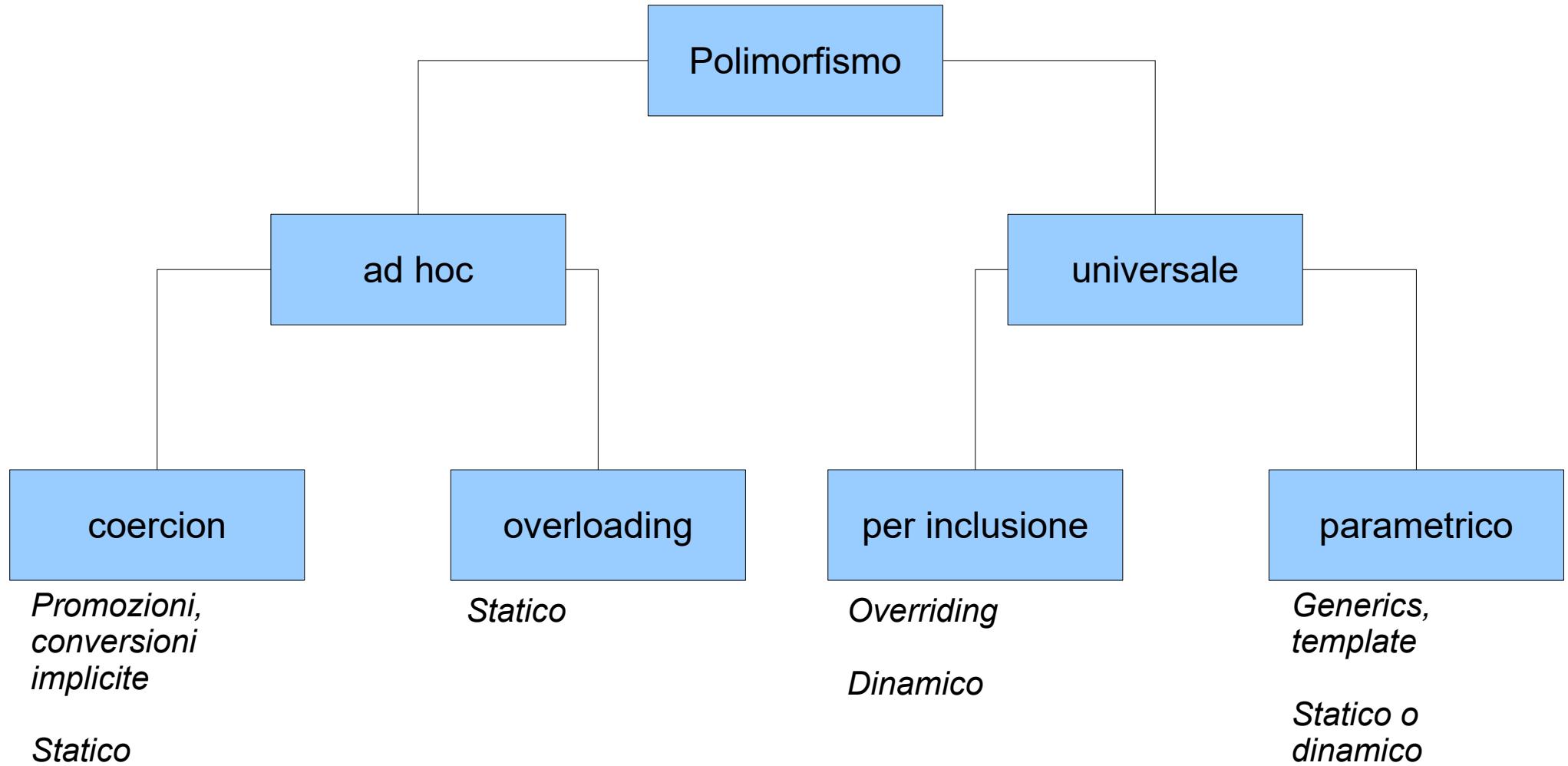
Definizione: Uno stesso oggetto sintattico (funzione, metodo, variabile, etc.) appartiene a più tipi

Significato o comportamento diverso a seconda del contesto

Esempio: operatore + in Java

1 + 2 ha un tipo e un significato diverso da "a" + "b"

Polimorfismo



Polimorfismo

- **Ad hoc**
 - Specifico per particolari tipi di dato
 - compresi alcuni predefiniti
 - Si aggiungono singoli casi di polimorfismo ogni volta
- **Universale**
 - Si applica ad un numero di casi *illimitato a priori*
 - Ad esempio:
 - aggiungendo una nuova superclasse con un metodo concreto, l'overriding crea casi di polimorfismo rispetto a tutte le sottoclassi che già definiscono quel metodo
 - un template può definire una varietà di metodi di tipo diverso, il cui numero non è limitato a priori

Polimorfismo ad hoc

Overloading: *stesso nome ma diversa implementazione* a seconda dei tipi dei parametri

Esempio: overloading di operatori base

$23 + 4 : (\text{int}, \text{int}) : \text{int}$

$12.34 + 1.0 : (\text{double}, \text{double}) : \text{double}$

(internamente implementate con istruzioni diverse)

“Abc” + “dE3” : (`String`, `String`) : `String`

(concatenazione)

Polimorfismo ad hoc

Coercion: promozione automatica di tipi

Esempio: 12.34 + 1

<double> + <int>

converte <int> a <double>

converte 1 in 1.0

si riduce a <double> + <double>

Apparentemente, + ha *anche* il tipo (double,double) :
double

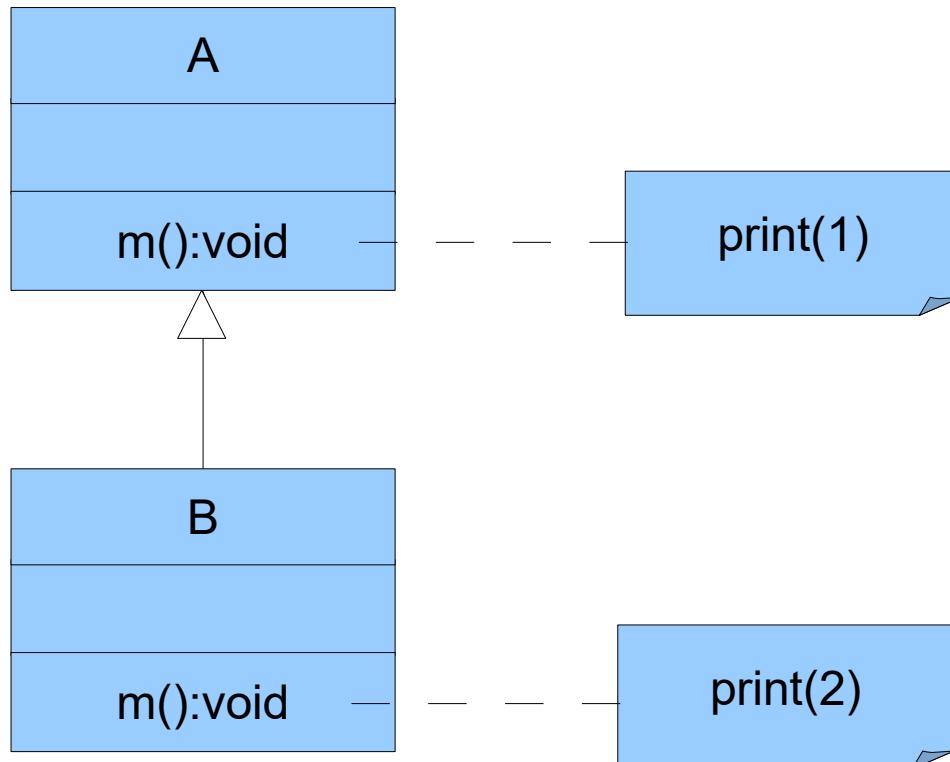
nonchè (int,double) : double

diverse implementazioni (le conversioni implicite
fanno parte dell'implementazione dell'istruzione)

Polimorfismo universale

Per inclusione

overriding nelle sottoclassi



A x; output

x = new A();

x.m(); 1

x = new B();

x.m(); 2

Polimorfismo universale

Parametrico

Tipi parametrici, generics o templates

Adatto a definire strutture omogenee

Ma si vedano le considerazioni sugli approcci ibridi
nelle slide di confronto tra polimorfismo per
inclusione e parametrico

Python

Tratti salienti

- Inventato da Guido van Rossum nel ~1990
- Ricco ecosistema di librerie e framework
- Linguaggio imperativo orientato agli oggetti
- Interpretato
- Dinamicamente tipizzato
- Offre *garbage collection*

Elementi di sintassi

```
n = int(input('Type a number, and its factorial will be printed: '))

if n < 0:
    raise ValueError('You must enter a non-negative integer')

factorial = 1

for i in range(2, n + 1):
    factorial *= i

print(factorial)
```

Osservazioni:

Sintassi snella

Non ci sono dichiarazioni di tipi
(tipico dei ling. di scripting)

L'indentazione è rilevante

Esecuzione

`python <nomefile>`

Oppure (in ambiente Unix):

iniziare il programma sorgente con:

`#!/bin/python`

Oppure invocare l'interprete come *read-eval-print loop* (REPL):

`python`

Numerose implementazioni alternative

Esempio: *mypy*, offre type checking statico

Elementi di semantica

- Tutti i valori sono oggetti
- Tutte le variabili sono riferimenti
- Come se in Java non esistessero i tipi primitivi, ma solo i wrapper
- Funzione **id(<exp>)**: restituisce l'indirizzo dell'oggetto puntato da <exp>
 - Ovvero, il valore del riferimento
- Funzione **type(<exp>)**: restituisce il tipo dell'oggetto puntato da <exp>

Modello dei dati e Data Objects

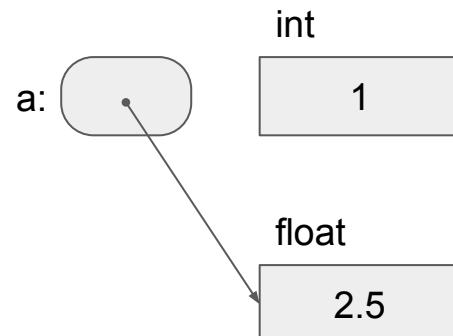
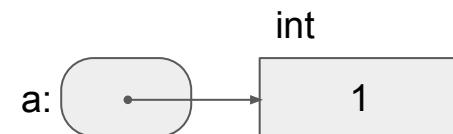
(Locazione, Nome, Valore, Tipo)

Codice Python:

```
a = 1  
id(a) → 140335477971640  
type(a) → type 'int'
```

```
a = 2.5  
id(a) → 573821909931  
type(a) → type 'float'
```

Memory layout:



Data Objects:

(loc1, a, loc2, "riferimento a int")
(loc2, anonimo, 1, int)
loc2 = 140335477971640

(loc1, a, loc3, "riferimento a float")
(loc3, anonimo, 2.5, float)
loc3 = 573821909931

Sistema dei tipi

Ricca varietà di tipi nativi:

- boolean
- *Numerici*: int, float, **complex**
- *Stringhe*: str, bytes, bytearray
- *Collezioni*: list, tuple, dict, set, frozenset

L'utente può creare nuove classi e funzioni

Le funzioni/metodi sono *first-class objects*

A proposito di interi

Il tipo *int* rappresenta interi di *dimensione arbitraria*

```
>>> 2**50  
1125899906842624
```

Esempio di scelta di *astrazione maggiore* rispetto a C, Java, etc.

Più flessibile, meno efficiente

In Java: classe standard BigInteger

In C e C++: librerie esterne

Per dettagli, si veda [Wikipedia: List of arbitrary-precision arithmetic software](#)

Mutabile o immutabile?

Mutabile vuol dire: cambia il valore, non cambia l'indirizzo (locazione)

Esercizi:

1. Scoprire empiricamente se il tipo **int** è mutabile
2. Scoprire empiricamente se il tipo **list** è mutabile

Type checking

I tipi non sono *dichiarati*, ma esistono e sono *controllati*

Esempi:

```
>>> 1+"a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
>>> a=1
>>> a.get(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'get'
```

Type checking

Solo se e quando il codice viene eseguito

Esempio:

```
a=1
```

```
if a==0:
```

```
b=1+"ciao"
```

Questo frammento viene eseguito senza problemi

Linguaggi di Programmazione I – Java-4

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

4 aprile 2023



Complementi

String e StringBuffer

Garbage collection

Questionario



String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

String e StringBuffer



String (1)

- Le stringhe sono oggetti che possono essere creati come segue:

```
String s = new String("abcdef");
```

String e
StringBuffer

String (1)
Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



String (1)

- Le stringhe sono oggetti che possono essere creati come segue:

```
String s = new String("abcdef");
```

oppure così:

```
String s = "abcdef";
```

- Vedremo tra poco quali sono le (sottili) differenze tra questi modi.

[String e
StringBuffer](#)

[String \(1\)](#)

Immutabilità

Esempi

String pool

[StringBuffer \(1\)](#)

Uguaglianza

[Garbage collection](#)

[Questionario](#)



String (1)

- Le stringhe sono oggetti che possono essere creati come segue:

```
String s = new String("abcdef");
```

oppure così:

```
String s = "abcdef";
```

- Vedremo tra poco quali sono le (sottili) differenze tra questi modi.

[String e
StringBuffer](#)

[String \(1\)](#)

Immutabilità

Esempi

String pool

[StringBuffer \(1\)](#)

Uguaglianza

[Garbage collection](#)

[Questionario](#)



String (1)

- Le stringhe sono oggetti che possono essere creati come segue:

```
String s = new String("abcdef");
```

oppure così:

```
String s = "abcdef";
```

- Vedremo tra poco quali sono le (sottili) differenze tra questi modi.

- Nota: "abcdef" si chiama *letterale String*
- Analogamente, 1.0 si chiama *letterale double*

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



Immutabilità

- Le stringhe sono oggetti **immutabili**
- Immutabilità significa che, una volta assegnato all'oggetto un contenuto, esso è fissato per sempre (capiremo tra poco anche perché deve essere così).

String e

StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



Immutabilità

- Le stringhe sono oggetti **immutabili**
- Immutabilità significa che, una volta assegnato all'oggetto un contenuto, esso è fissato per sempre (capiremo tra poco anche perché deve essere così).
- Attenzione: immutabili sono gli oggetti String, non i loro riferimenti. Questi ultimi possono cambiare valore.
- Anche i tipi wrapper sono immutabili

String e

StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



Esempi

[String e
StringBuffer](#)

[String \(1\)](#)

[Immutabilità](#)

[Esempi](#)

[String pool](#)

[StringBuffer \(1\)](#)

[Uguaglianza](#)

[Garbage collection](#)

[Questionario](#)

```
String s = "Walter";
```



Esempi

[String e
StringBuffer](#)

[String \(1\)](#)

[Immutabilità](#)

[Esempi](#)

[String pool](#)

[StringBuffer \(1\)](#)

[Uguaglianza](#)

[Garbage collection](#)

[Questionario](#)

```
String s = "Walter";
String s2 = s;
```



Esempi

[String e
StringBuffer](#)

[String \(1\)
Immutabilità](#)

[Esempi](#)

[String pool](#)

[StringBuffer \(1\)](#)

[Uguaglianza](#)

[Garbage collection](#)

[Questionario](#)

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
```



Esempi

String e

StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
```



Esempi

[String e](#)

[StringBuffer](#)

[String \(1\)](#)

[Immutabilità](#)

[Esempi](#)

[String pool](#)

[StringBuffer \(1\)](#)

[Uguaglianza](#)

[Garbage collection](#)

[Questionario](#)

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
x.concat(" White");
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
x.concat(" White");
System.out.println(x);    // cosa stampa?
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
x.concat(" White");
System.out.println(x);    // cosa stampa?
```

```
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";                // Quanti oggetti in gioco?
System.out.println(s1 + s2); // Cosa stampa?
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
x.concat(" White");
System.out.println(x);    // cosa stampa?
```

```
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";                // Quanti oggetti in gioco?
System.out.println(s1 + s2); // Cosa stampa?
```

```
String s1 = "abc";
String s2 = s1 + "";
String s3 = "abc";
System.out.println(s1 == s2);
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
x.concat(" White");
System.out.println(x);    // cosa stampa?
```

```
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";                // Quanti oggetti in gioco?
System.out.println(s1 + s2); // Cosa stampa?
```

```
String s1 = "abc";
String s2 = s1 + "";
String s3 = "abc";
System.out.println(s1 == s2); // false!
System.out.println(s1 == s3);
```



Esempi

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario

```
String s = "Walter";
String s2 = s;
s = s.concat(" White");
System.out.println(s);    // cosa stampa?
System.out.println(s2);   // e qui?
```

```
String x = "Walter";
x.concat(" White");
System.out.println(x);    // cosa stampa?
```

```
String s1 = "A";
String s2 = s1 + "B";
s1.concat("C");
s2.concat(s1);
s1 += "D";                // Quanti oggetti in gioco?
System.out.println(s1 + s2); // Cosa stampa?
```

```
String s1 = "abc";
String s2 = s1 + "";
String s3 = "abc";
System.out.println(s1 == s2); // false!
System.out.println(s1 == s3); // true!
```



String pool

- Per motivi di efficienza, poichè nelle applicazioni i letterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



String pool

- Per motivi di efficienza, poichè nelle applicazioni i letterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.
- Quando il compilatore incontra un letterale `String`, esso controlla se è già presente nel pool.

String e

StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



String pool

- Per motivi di efficienza, poichè nelle applicazioni i letterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.
- Quando il compilatore incontra un letterale `String`, esso controlla se è già presente nel pool.
 - ◆ Se è presente, allora il letterale viene interpretato come un riferimento all'oggetto `String` esistente
 - ◆ Altrimenti, viene creato un nuovo oggetto `String` e aggiunto al pool.

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



String pool

- Per motivi di efficienza, poichè nelle applicazioni i letterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.
- Quando il compilatore incontra un letterale `String`, esso controlla se è già presente nel pool.
 - ◆ Se è presente, allora il letterale viene interpretato come un riferimento all'oggetto `String` esistente
 - ◆ Altrimenti, viene creato un nuovo oggetto `String` e aggiunto al pool.
- Questo meccanismo di condivisione può funzionare perché gli oggetti `String` sono immutabili.

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



String pool

- Per motivi di efficienza, poichè nelle applicazioni i letterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.
- Quando il compilatore incontra un letterale `String`, esso controlla se è già presente nel pool.
 - ◆ Se è presente, allora il letterale viene interpretato come un riferimento all'oggetto `String` esistente
 - ◆ Altrimenti, viene creato un nuovo oggetto `String` e aggiunto al pool.
- Questo meccanismo di condivisione può funzionare perché gli oggetti `String` sono immutabili. Se lo stesso letterale compare in punti diversi del codice, l'eventuale modifica di un letterale modificherebbe anche l'altro

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



String pool

- Per motivi di efficienza, poichè nelle applicazioni i letterali `String` occupano molta memoria, la JVM riserva un'area speciale di memoria ad essi: la *String constant pool*.
- Quando il compilatore incontra un letterale `String`, esso controlla se è già presente nel pool.
 - ◆ Se è presente, allora il letterale viene interpretato come un riferimento all'oggetto `String` esistente
 - ◆ Altrimenti, viene creato un nuovo oggetto `String` e aggiunto al pool.
- Questo meccanismo di condivisione può funzionare perché gli oggetti `String` sono immutabili. Se lo stesso letterale compare in punti diversi del codice, l'eventuale modifica di un letterale modificherebbe anche l'altro
- Qual è quindi la differenza tra i due enunciati?

```
String s = "abcdef";
```

```
String s = new String("abcdef");
```



StringBuffer (1)

- Se si deve fare un uso intensivo di manipolazione di stringhe, allora è opportuno usare le classi `StringBuilder` e `StringBuffer`: esse sono simili a `String`, ma sono *mutabili*
- La differenza tra le due è che `StringBuffer` è *thread-safe*

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



StringBuffer (1)

- Se si deve fare un uso intensivo di manipolazione di stringhe, allora è opportuno usare le classi `StringBuilder` e `StringBuffer`: esse sono simili a `String`, ma sono *mutabili*
- La differenza tra le due è che `StringBuffer` è *thread-safe*
- Per esempio:

```
StringBuffer s = new StringBuffer("Walter");
s.append(" White");
System.out.println(s); // cosa stampa?

String contenuto = s.toString();
```

String e
StringBuffer

String (1)

Immutabilità

Esempi

String pool

StringBuffer (1)

Uguaglianza

Garbage collection

Questionario



StringBuffer (1)

- Se si deve fare un uso intensivo di manipolazione di stringhe, allora è opportuno usare le classi `StringBuilder` e `StringBuffer`: esse sono simili a `String`, ma sono *mutabili*
- La differenza tra le due è che `StringBuffer` è *thread-safe*
- Per esempio:

```
StringBuffer s = new StringBuffer("Walter");
s.append(" White");
System.out.println(s); // cosa stampa?

String contenuto = s.toString();
```

- Attenzione:

```
StringBuffer s = "abc";
// Illegale: String e StringBuffer
// non sono auto-convertibili!
```



StringBuffer (1)

- Se si deve fare un uso intensivo di manipolazione di stringhe, allora è opportuno usare le classi `StringBuilder` e `StringBuffer`: esse sono simili a `String`, ma sono *mutabili*
- La differenza tra le due è che `StringBuffer` è *thread-safe*
- Per esempio:

```
StringBuffer s = new StringBuffer("Walter");
s.append(" White");
System.out.println(s); // cosa stampa?

String contenuto = s.toString();
```

- Attenzione:

```
StringBuffer s = "abc";
// Illegale: String e StringBuffer
// non sono auto-convertibili!

StringBuffer s = (StringBuffer) "abc";
// Illegale: neanche con cast!
```



Uguaglianza

- ATTENZIONE: mentre la classe `String` sovrascrive il metodo `equals` in modo da controllare l'uguaglianza del contenuto dei due oggetti (quello corrente e quello ricevuto come parametro), le classi `StringBuffer` e `StringBuilder` non lo sovrascrivono ed usano quello ereditato da `Object`, che funziona come l'operatore `==` (cioè compara i riferimenti).

[String e](#)

[StringBuffer](#)

[String \(1\)](#)

[Immutabilità](#)

[Esempi](#)

[String pool](#)

[StringBuffer \(1\)](#)

[Uguaglianza](#)

[Garbage collection](#)

[Questionario](#)



Uguaglianza

- ATTENZIONE: mentre la classe `String` sovrascrive il metodo `equals` in modo da controllare l'uguaglianza del contenuto dei due oggetti (quello corrente e quello ricevuto come parametro), le classi `StringBuffer` e `StringBuilder` non lo sovrascrivono ed usano quello ereditato da `Object`, che funziona come l'operatore `==` (cioè compara i riferimenti).
- Le classi `String`, `StringBuffer` e `StringBuilder` sono final. Esse non possono essere specializzate: sovrascriverne i metodi potrebbe creare problemi di sicurezza.



String e
StringBuffer

Garbage collection

Definizioni

Algoritmo di GC

Esempio (2)

Esempio (3)

Esempio (4)

Esempio (5)

Questionario

Garbage collection



Definizioni

- Rilascio automatico della memoria non più accessibile dal programma
- Un data object è “eleggibile” per la GC se non è più accessibile dal programma

Esempio:

```
public static void main(String[] args) {  
    StringBuffer sb = new StringBuffer("Ciao");  
    System.out.println(sb);  
    sb = null;  
    // ora l'oggetto StringBuffer e' eleggibile per la GC  
}
```

Nota: tranne casi eccezionali, le stringhe nello string pool non sono mai eleggibili per la GC



Algoritmo di GC

[String e
StringBuffer](#)

[Garbage collection](#)

Definizioni

Algoritmo di GC

Esempio (2)

Esempio (3)

Esempio (4)

Esempio (5)

[Questionario](#)

Algoritmo *mark-and-sweep*

1. Fase *mark*: a partire dallo stack e dalla regione statica, esplorare tutti gli oggetti accessibili e marcarli come tali
2. Fase *sweep*: rilasciare tutti i data object dello heap che non sono stati marcati come accessibili

La fase mark è analoga alla visita di un grafo: i nodi sono oggetti e gli archi sono riferimenti tra oggetti

Nota: la JVM contiene diversi algoritmi di GC alternativi



Esempio (2)

```
public static void main(String[] args) {  
    StringBuffer s1 = new StringBuffer("Ciao");  
    StringBuffer s2 = new StringBuffer("Addio");  
    System.out.println(s1);  
    // l'oggetto riferito da s1 non e' ancora  
    // eleggibile per GC
```

String e
StringBuffer

Garbage collection

Definizioni

Algoritmo di GC

Esempio (2)

Esempio (3)

Esempio (4)

Esempio (5)

Questionario



Esempio (2)

```
public static void main(String[] args) {  
    StringBuffer s1 = new StringBuffer("Ciao");  
    StringBuffer s2 = new StringBuffer("Addio");  
    System.out.println(s1);  
    // l'oggetto riferito da s1 non e' ancora  
    // eleggibile per GC  
    s1 = s2;  
    // qui e' eleggibile  
    ...  
}
```

String e
StringBuffer

Garbage collection

Definizioni

Algoritmo di GC

Esempio (2)

Esempio (3)

Esempio (4)

Esempio (5)

Questionario



Esempio (3)

String e
StringBuffer

Garbage collection

Definizioni

Algoritmo di GC

Esempio (2)

Esempio (3)

Esempio (4)

Esempio (5)

Questionario

```
1. import java.util.Date;
2. public class TestGC {
3.     public static void main(String[] args) {
4.         Date d = getDate();
5.         System.out.println(d);
6.     }
7.
8.     public static Date getDate() {
9.         Date d2 = new Date();
10.        String now = d2.toString();
11.        System.out.println(now);
12.        return d2;
13.    }
14. }
```

Tracciare il codice con riferimento alla eleggibilità per GC degli oggetti

(Simulare l'esecuzione e indicare quali oggetti sono eleggibili ad ogni passo)



Esempio (4)

```
public class Isola {  
    Isola i;  
  
    public static void main(String[] args) {  
        Isola i1 = new Isola();  
        Isola i2 = new Isola();  
        Isola i3 = new Isola();  
  
        i1.i = i2;  
        i2.i = i3;  
        i3.i = i1;  
  
        i1 = null;  
        i2 = null;  
        i3 = null;  
  
        // qui quali oggetti sono eleggibili?  
    }  
}
```



Esempio (5)

Simulare la procedura di GC mark-and-sweep nel punto segnato nel seguente programma:

```
class Employee {  
    private String name;  
    private Employee boss;  
    public final static Employee CEO = new Employee("Gustavo", null);  
    ...  
  
    public static void main(String[] args) {  
        Employee w = new Employee("Walter", CEO);  
        f();  
    }  
    public static void f() {  
        Employee j = new Employee("Jesse", CEO),  
              p = new Employee("Pete", j);  
        ArrayList<Employee> l = new ArrayList<>();  
        l.add(j);  
        j = null;  
        p = null;  
        // simulare la GC a questo punto  
    }  
}
```



String e

StringBuffer

Garbage collection

Questionario

D 1

D 2

D 3

Questionario



D 1

Data la stringa costruita mediante `s = new String("xyzzy")`, quali delle seguenti invocazioni di metodi modificherà la stringa?

- A. `s.append("aaa");`
- B. `s.trim();`
- C. `s.substring(3);`
- D. `s.replace('z', 'a');`
- E. `s.concat(s);`
- F. Nessuna delle precedenti.

[String e
StringBuffer](#)

[Garbage collection](#)

[Questionario](#)

D 1

[D 2](#)

[D 3](#)



D 1

Data la stringa costruita mediante `s = new String("xyzzy")`, quali delle seguenti invocazioni di metodi modificherà la stringa?

- A. `s.append("aaa")`;
- B. `s.trim()`;
- C. `s.substring(3)`;
- D. `s.replace('z', 'a')`;
- E. `s.concat(s)`;
- F. Nessuna delle precedenti.

F. – Gli oggetti `String` sono immutabili.



D 2

Qual è l'output del seguente brano di codice?

```
1. String s1 = "abc" + "def";  
2. String s2 = new String(s1);  
3. if (s1 == s2)  
4.     System.out.println("A");  
5. if (s1.equals(s2))  
6.     System.out.println("B");
```

- A. AB
 - B. A
 - C. B
 - D. Nessun output.
-



D 2

Qual è l'output del seguente brano di codice?

```
1. String s1 = "abc" + "def";  
2. String s2 = new String(s1);  
3. if (s1 == s2)  
4.     System.out.println("A");  
5. if (s1.equals(s2))  
6.     System.out.println("B");
```

- A. AB
- B. A
- C. B
- D. Nessun output.

C.



D 3

Quanti oggetti sono prodotti nel seguente frammento di codice?

```
1. StringBuffer sbuf = new StringBuffer("abcde");  
2. sbuf.insert(3, "xyz");
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

String e

StringBuffer

Garbage collection

Questionario

D 1

D 2

D 3



D 3

Quanti oggetti sono prodotti nel seguente frammento di codice?

```
1. StringBuffer sbuf = new StringBuffer("abcde");  
2. sbuf.insert(3, "xyz");
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

C.

Linguaggi di Programmazione I – Lezione 15

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

3 aprile 2023



Eccezioni (Gestione degli errori)

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario



**Eccezioni: il
meccanismo**

Introduzione
Meccanismi
linguistici

Lanciare
un'eccezione
Ciclo di vita di
un'eccezione

Catturare le
eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

Eccezioni: il meccanismo



Introduzione

[Eccezioni: il meccanismo](#)

[Introduzione](#)
Meccanismi
linguistici

Lanciare
un'eccezione
Ciclo di vita di
un'eccezione

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- Le eccezioni denotano “eventi eccezionali” la cui occorrenza altera il flusso normale delle istruzioni



Introduzione

[Eccezioni: il meccanismo](#)

[Introduzione](#)
Meccanismi
linguistici

[Lanciare
un'eccezione](#)
Ciclo di vita di
un'eccezione

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- Le eccezioni denotano “eventi eccezionali” la cui occorrenza altera il flusso normale delle istruzioni
- Es.: risorse hardware indisponibili, hardware malfunzionante, bachi nel software ...



Introduzione

[Eccezioni: il meccanismo](#)

Introduzione

Meccanismi linguistici

Lanciare un'eccezione

Ciclo di vita di un'eccezione

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- Le eccezioni denotano “eventi eccezionali” la cui occorrenza altera il flusso normale delle istruzioni
- Es.: risorse hardware indisponibili, hardware malfunzionante, bachi nel software ...
- Quando capita un tale evento, si dice che viene “lanciata una eccezione”



Meccanismi linguistici

[Eccezioni: il meccanismo](#)

[Introduzione
Meccanismi
linguistici](#)

[Lanciare
un'eccezione
Ciclo di vita di
un'eccezione](#)

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

Il linguaggio supporta i seguenti meccanismi relativi alle eccezioni:

- **Lanciare** un'eccezione (istruzione `throw`)
- **Dichiarare** che un metodo lancia un'eccezione (dichiarazione `throws`)
- **Catturare** un'eccezione (blocco `try-catch`)



Lanciare un'eccezione

- In Java tutto ciò che non è primitivo è un oggetto.

[Eccezioni: il meccanismo](#)

Introduzione
Meccanismi
linguistici

**Lanciare
un'eccezione**

Ciclo di vita di
un'eccezione

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Lanciare un'eccezione

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno “eccezione” a questa regola.

[Eccezioni: il meccanismo](#)

[Introduzione
Meccanismi
linguistici](#)

**[Lanciare
un'eccezione](#)**

[Ciclo di vita di
un'eccezione](#)

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Lanciare un'eccezione

[Eccezioni: il meccanismo](#)

Introduzione
Meccanismi
linguistici

**Lanciare
un'eccezione**

Ciclo di vita di
un'eccezione

Catturare le
eccezioni

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno “eccezione” a questa regola.
- Ogni eccezione è una istanza di una sottoclasse della classe `Throwable`



Lanciare un'eccezione

[Eccezioni: il meccanismo](#)

Introduzione
Meccanismi
linguistici

**Lanciare
un'eccezione**

Ciclo di vita di
un'eccezione

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- In Java tutto ciò che non è primitivo è un oggetto. Le eccezioni non fanno “eccezione” a questa regola.
- Ogni eccezione è una istanza di una sottoclasse della classe `Throwable`
- Una eccezione viene lanciata usando la parola riservata `throw`:

```
throw <exp>;
```

dove `<exp>` è un'espressione di tipo dichiarato `Throwable` o suo sottotipo.

Esempio:

```
throw new IllegalArgumentException();
```



Ciclo di vita di un'eccezione

[Eccezioni: il meccanismo](#)

[Introduzione
Meccanismi
linguistici](#)

[Lanciare
un'eccezione](#)

**[Ciclo di vita di
un'eccezione](#)**

[Catturare le
eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- Lanciare un'eccezione interrompe il normale flusso di esecuzione
- Se non viene *catturata* localmente, l'eccezione termina il metodo corrente e passa al chiamante, che ha la possibilità di catturarla
- Se neanche il metodo chiamante la cattura, l'eccezione continua a risalire lo stack di attivazione, fino a raggiungere il main
- Se neanche il main la cattura, l'eccezione termina il programma e la JVM stampa il contenuto dell'eccezione (*stack trace*)



[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[try e catch](#)

[finally](#)

[Vincoli sintattici](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

Catturare le eccezioni



try e catch

- Per catturare le eccezioni, il codice che potrebbe lanciare eccezioni viene inglobato in un blocco marcato `try`
- Il codice che assume la responsabilità di gestire un'eccezione va inglobato in una clausola `catch`

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[try e catch](#)

[finally](#)

[Vincoli sintattici](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



try e catch

- Per catturare le eccezioni, il codice che potrebbe lanciare eccezioni viene inglobato in un blocco marcato `try`
- Il codice che assume la responsabilità di gestire un'eccezione va inglobato in una clausola `catch`

```
try {  
    // Codice "rischioso"  
} catch (Eccezione1 e) {  
    // Codice che gestisce una Eccezione1  
} catch (Eccezione2 e) {  
    // Codice che gestisce una Eccezione2  
}  
// Codice non rischioso
```

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[try e catch](#)

[finally](#)

[Vincoli sintattici](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



finally

- Un blocco (opzionale) marcato finally verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {  
    // Codice "rischioso"  
} catch (Eccezione1 e) {  
    // Codice che gestisce una Eccezione1  
} catch (Eccezione2 e) {  
    // Codice che gestisce una Eccezione2  
} finally {  
    // Codice da eseguire in ogni caso  
}  
// Codice non rischioso
```



finally

- Un blocco (opzionale) marcato finally verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {  
    // Codice "rischioso"  
} catch (Eccezione1 e) {  
    // Codice che gestisce una Eccezione1  
} catch (Eccezione2 e) {  
    // Codice che gestisce una Eccezione2  
} finally {  
    // Codice da eseguire in ogni caso  
}  
// Codice non rischioso
```

- Il blocco finally viene eseguito perfino dopo una eventuale istruzione return presente nei blocchi try o catch



finally

- Un blocco (opzionale) marcato finally verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {  
    // Codice "rischioso"  
} catch (Eccezione1 e) {  
    // Codice che gestisce una Eccezione1  
} catch (Eccezione2 e) {  
    // Codice che gestisce una Eccezione2  
} finally {  
    // Codice da eseguire in ogni caso  
}  
// Codice non rischioso
```

- Il blocco finally viene eseguito perfino dopo una eventuale istruzione return presente nei blocchi try o catch
- IL BLOCCO finally VIENE ESEGUITO SEMPRE



finally

- Un blocco (opzionale) marcato finally verrà SEMPRE eseguito, anche dopo il lancio e la gestione (eventuale) dell'eccezione.

```
try {  
    // Codice "rischioso"  
} catch (Eccezione1 e) {  
    // Codice che gestisce una Eccezione1  
} catch (Eccezione2 e) {  
    // Codice che gestisce una Eccezione2  
} finally {  
    // Codice da eseguire in ogni caso  
}  
// Codice non rischioso
```

- Il blocco finally viene eseguito perfino dopo una eventuale istruzione return presente nei blocchi try o catch
- IL BLOCCO finally VIENE ESEGUITO SEMPRE
- Il blocco finally potrebbe non essere eseguito o potrebbe non completare l'esecuzione solo in conseguenza di un crash totale del sistema oppure tramite una invocazione di System.exit(int status)



Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)
[try e catch](#)
[finally](#)

Vincoli sintattici

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Vincoli sintattici

- Le clausole catch ed il blocco finally sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco try solitario causa un errore di compilazione
- Se esistono una o più clausole catch, esse devono seguire immediatamente il blocco try

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[try e catch
finally](#)

Vincoli sintattici

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione
- Se esistono una o più clausole `catch`, esse devono seguire immediatamente il blocco `try`
- Se esiste il blocco `finally`, esso deve comparire per ultimo

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)
[try e catch](#)
[finally](#)

Vincoli sintattici

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Vincoli sintattici

- Le clausole `catch` ed il blocco `finally` sono opzionali, ma deve essere presente **almeno uno dei due**
- Un blocco `try` solitario causa un errore di compilazione
- Se esistono una o più clausole `catch`, esse devono seguire immediatamente il blocco `try`
- Se esiste il blocco `finally`, esso deve comparire per ultimo
- È significativo l'ordine delle clausole `catch` (vedremo tra poco)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[try e catch finally](#)

Vincoli sintattici

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[*Handle or Declare*](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

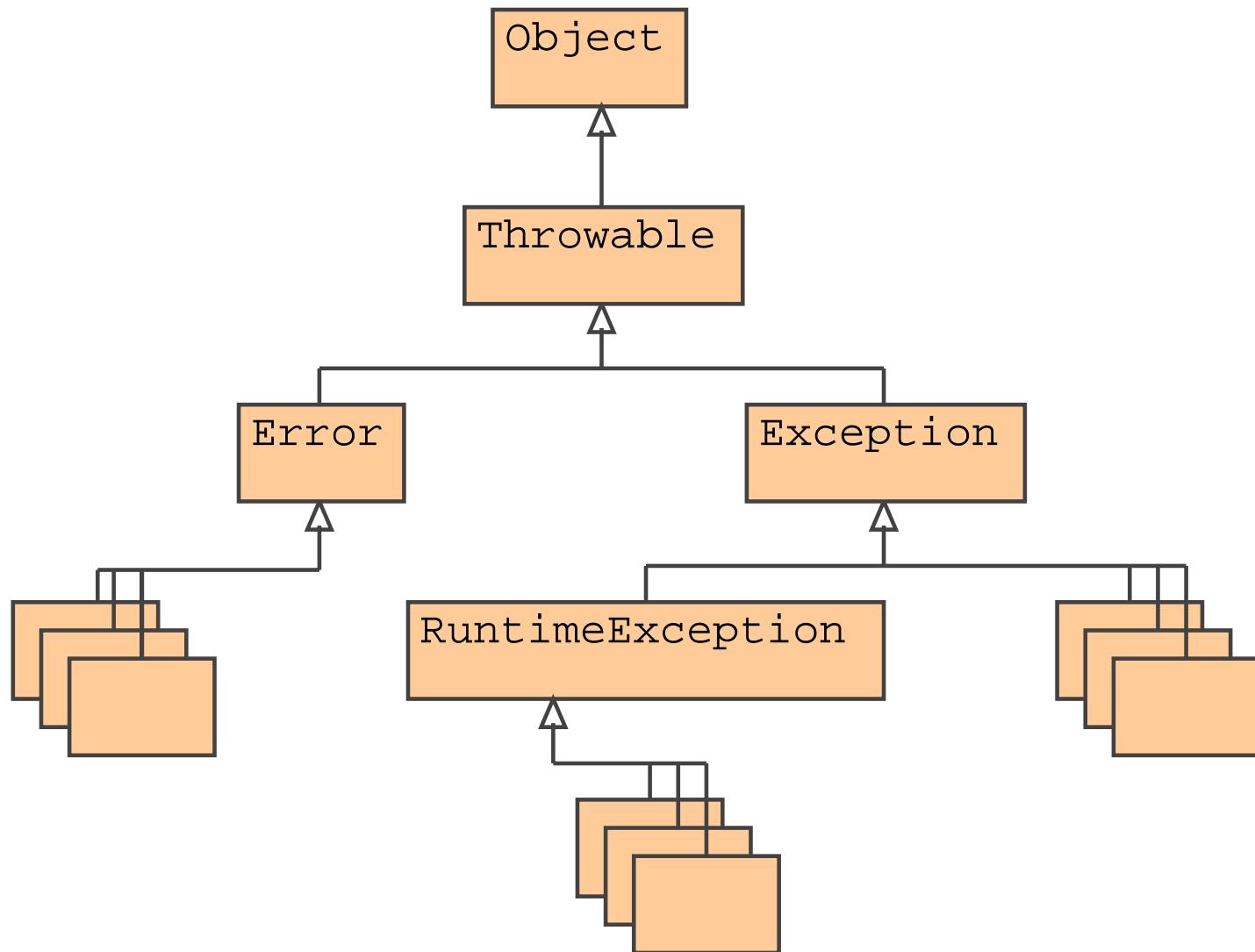
[Questionario](#)

Eccezioni: i dettagli



Gerarchia (1)

Ecco le principali classi di eccezioni:



[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

Gerarchia (1)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

Gerarchia (2)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

Handle or Declare

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati. Essa contiene il metodo `printStackTrace`.
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)
[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati. Essa contiene il metodo `printStackTrace`.
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile.
In genere, una applicazione non è capace di riprendersi da una situazione di errore. Pertanto, queste eccezioni solitamente non vengono catturate.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)
[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Gerarchia (2)

- La classe `Throwable` rappresenta tutti gli oggetti che possono essere lanciati. Essa contiene il metodo `printStackTrace`.
- La classe `Error` e le sue sottoclassi rappresentano situazioni insolite che non sono causate da errori di programmazione o da ciò che normalmente succede durante l'esecuzione del programma. Per esempio, la JVM ha esaurito la memoria oppure qualche altra risorsa non è disponibile.
In genere, una applicazione non è capace di riprendersi da una situazione di errore. Pertanto, queste eccezioni solitamente non vengono catturate.
- La classe `RuntimeException` rappresenta pure eventi eccezionali, ma dovuti al programma (errori di programmazione, bachi). Il programmatore che si accorge di un baco dovuto ad un suo errore deve correggerlo, non gestirlo! Pertanto, anche queste eccezioni solitamente non vengono catturate.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

Gerarchia (2)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Eccezioni catturate (1)

- Una clausola `catch (E e)` cattura ogni oggetto-eccezione il cui tipo effettivo è sottotipo di `E`

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[*Handle or Declare*](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Eccezioni catturate (1)

- Una clausola `catch (E e)` cattura ogni oggetto-eccezione il cui tipo effettivo è sottotipo di `E`
- Un'eccezione di tipo effettivo `E` verrà catturata dal *primo* blocco `catch` in grado di catturarla
- Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Eccezioni catturate (1)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- Una clausola `catch (E e)` cattura ogni oggetto-eccezione il cui tipo effettivo è sottotipo di `E`
- Un'eccezione di tipo effettivo `E` verrà catturata dal *primo* blocco `catch` in grado di catturarla
- Esempio: la classe `IndexOutOfBoundsException` ha due sottoclassi, `ArrayIndexOutOfBoundsException` e `StringIndexOutOfBoundsException`; si può scrivere una unica clausola che catturi una qualunque di queste eccezioni:

```
try {  
    // Codice che potrebbe lanciare una eccezione  
    // IndexOutOfBoundsException oppure  
    // ArrayIndexOutOfBoundsException oppure  
    // StringIndexOutOfBoundsException  
}  
catch (IndexOutOfBoundsException e) {  
    e.printStackTrace();  
}
```



Casi particolari

- Eventuali eccezioni lanciate dall'interno dei blocchi catch e finally non vengono catturate dagli altri blocchi catch dello stesso costrutto
- Se proprio necessario, i try-catch possono essere annidati

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

Casi particolari

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

Handle or Declare

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Eccezioni catturate (2)

■ Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {  
    // codice rischioso  
} catch (Exception e) {  
}
```

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[*Handle or Declare*](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Eccezioni catturate (2)

- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {  
    // codice rischioso  
} catch (Exception e) {  
}
```

- L'ordine delle clausole catch è importante.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Eccezioni catturate (2)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- Resistere alla tentazione di scrivere una unica clausola catch-all:

```
try {  
    // codice rischioso  
} catch (Exception e) {  
}
```

- L'ordine delle clausole catch è importante.
- Nell'esempio precedente, se avessimo scritto:

```
try {  
    // Codice che potrebbe lanciare una eccezione  
    //     IndexOutOfBoundsException, oppure  
    //     ArrayIndexOutOfBoundsException  
} catch (IndexOutOfBoundsException e) {  
    // Gestisce l'eccezione  
} catch (ArrayIndexOutOfBoundsException e) {  
    // Gestisce l'eccezione  
}
```

il codice non sarebbe stato compilato, perché il secondo catch è ridondante



Eccezioni catturate (3)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- È corretto, invece, scrivere:

```
try {  
    // Codice che potrebbe lanciare una eccezione  
    //     IndexOutOfBoundsException, oppure  
    //     ArrayIndexOutOfBoundsException  
} catch (ArrayIndexOutOfBoundsException e) {  
    // Gestisce l'eccezione  
} catch (IndexOutOfBoundsException e) {  
    // Gestisce l'eccezione  
}
```



... e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

Handle or Declare

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



... e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di Error o di RuntimeException).

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

Handle or Declare

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



... e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di Error o di RuntimeException).
- La parola chiave throws viene usata per elencare le eccezioni che possono fuoriuscire da un metodo:

```
void miaFunzione() throws MiaEccezione1, MiaEccezione2 {  
    // qui il codice per il metodo  
}
```

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



... e non catturate

- Come facciamo a sapere che un metodo può lanciare una eccezione che dobbiamo catturare?
- Così come la dichiarazione del metodo deve specificare il numero e il tipo dei parametri, il tipo di ritorno, anche le eccezioni che un metodo può lanciare DEVONO essere dichiarate (a meno che non siano sottoclassi di Error o di RuntimeException).
- La parola chiave throws viene usata per elencare le eccezioni che possono fuoriuscire da un metodo:

```
void miaFunzione() throws MiaEccezione1, MiaEccezione2 {  
    // qui il codice per il metodo  
}
```

- Il fatto che un metodo dichiara l'eccezione non significa che esso la lancerà sempre, ma avverte l'utilizzatore che esso potrebbe lanciarla.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
 1. Gestire l'eccezione fornendo le opportune sezioni try/catch

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
 1. Gestire l'eccezione fornendo le opportune sezioni try/catch
 2. Dichiarare l'eccezione nell'intestazione del metodo

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
 1. Gestire l'eccezione fornendo le opportune sezioni try/catch
 2. Dichiarare l'eccezione nell'intestazione del metodo
- Una “eccezione” a questa regola: Error e RuntimeException sono esenti dall'obbligo di dichiarazione

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
 1. Gestire l'eccezione fornendo le opportune sezioni try/catch
 2. Dichiarare l'eccezione nell'intestazione del metodo
- Una “eccezione” a questa regola: Error e RuntimeException sono esenti dall’obbligo di dichiarazione
Esse sono *unchecked* (non verificate) dal compilatore, mentre le rimanenti eccezioni sono dette *checked* (verificate)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
 1. Gestire l'eccezione fornendo le opportune sezioni try/catch
 2. Dichiarare l'eccezione nell'intestazione del metodo
- Una “eccezione” a questa regola: Error e RuntimeException sono esenti dall'obbligo di dichiarazione
Esse sono *unchecked* (non verificate) dal compilatore, mentre le rimanenti eccezioni sono dette *checked* (verificate)
Ora, forse, si capisce il motivo della gerarchia precedentemente esposta.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Handle or Declare

- Se un metodo non lancia direttamente una eccezione, ma richiama un altro metodo che può farlo, allora si deve scegliere almeno una di queste opzioni (regola *handle or declare*):
 1. Gestire l'eccezione fornendo le opportune sezioni try/catch
 2. Dichiarare l'eccezione nell'intestazione del metodo
- Una “eccezione” a questa regola: Error e RuntimeException sono esenti dall'obbligo di dichiarazione
Esse sono *unchecked* (non verificate) dal compilatore, mentre le rimanenti eccezioni sono dette *checked* (verificate)
Ora, forse, si capisce il motivo della gerarchia precedentemente esposta.
- Nota: l'*or* della regola non è esclusivo, nel senso che si può decidere di fare entrambe le cose.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Esempio (1)

Quali problemi ci sono in questo codice?

```
void f1() {  
    f2();  
}  
  
void f2() {  
    throw new IOException();  
}
```

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[*Handle or Declare*](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Esempio (1)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

Quali problemi ci sono in questo codice?

```
void f1() {  
    f2();  
}  
  
void f2() {  
    throw new IOException();  
}
```

- Il metodo f2 lancia una eccezione checked ma non la dichiara; questo è un errore di compilazione



Esempio (1)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

Quali problemi ci sono in questo codice?

```
void f1() {  
    f2();  
}  
  
void f2() {  
    throw new IOException();  
}
```

- Il metodo f2 lancia una eccezione checked ma non la dichiara; questo è un errore di compilazione
- Se esso l'avesse dichiarata, come in:

```
void f2() throws IOException {...}
```

il problema l'avrebbe f1 che dovrebbe ora dichiararla o catturarla.



Esempio (2)

Quali problemi ci sono in questo codice?

```
import java.io.*;
class Test {
    public int f1() throws EOFException {
        return f2();
    }
    public int f2() throws EOFException {
        // qui il codice che lancia effettivamente l'eccezione
        return 1;
    }
}
```

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

Esempio (2)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Esempio (2)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

Esempio (2)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

Quali problemi ci sono in questo codice?

```
import java.io.*;
class Test {
    public int f1() throws EOFException {
        return f2();
    }
    public int f2() throws EOFException {
        // qui il codice che lancia effettivamente l'eccezione
        return 1;
    }
}
```

- Nessun problema
- Poiché EOFException è sottoclasse di IOException, che è sottoclasse di Exception, essa è una eccezione *checked*. Essa viene regolarmente dichiarata ed il codice regolarmente compilato.



Esempio (3)

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[*Handle or Declare*](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

Quali problemi ci sono in questo codice?

```
public void f1() {  
    // qui codice che puo' lanciare NullPointerException  
}
```



Esempio (3)

Quali problemi ci sono in questo codice?

```
public void f1() {  
    // qui codice che puo' lanciare NullPointerException  
}
```

- Nessun problema
- Poiché NullPointerException è sottoclasse di RuntimeException, essa è una eccezione *unchecked*. Non è necessario nè dichiararla, nè catturarla, ed il codice viene regolarmente compilato.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Esempio (4)

Analogamente, questo codice compila correttamente:

```
class TestEx {  
    public static void main (String [] args) {  
        mioMetodo();  
    }  
    static void mioMetodo() { // Non c'e' bisogno di  
                           // dichiarare un Error  
        fai();  
    }  
    static void fai() { // Non c'e' bisogno di dichiarare un Error  
        try {  
            throw new Error();  
        }  
        catch(Error me) {  
            throw me; // Ti ho preso, ma ora di rilancio  
        }  
    }  
}
```

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Nuove eccezioni

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception { }
```

oppure estendendo una qualunque sottoclasse di `Exception`.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

Handle or Declare

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Nuove eccezioni

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception { }
```

oppure estendendo una qualunque sottoclasse di `Exception`.

- Da questo momento in poi si può lanciare un oggetto del tipo (checked) `MiaEccezione`.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

Handle or Declare

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)



Nuove eccezioni

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Gerarchia \(1\)](#)

[Gerarchia \(2\)](#)

[Eccez. catturate \(1\)](#)

[Casi particolari](#)

[Eccez. catturate \(2\)](#)

[Eccez. catturate \(3\)](#)

[... e non catturate](#)

[Handle or Declare](#)

[Esempio \(1\)](#)

[Esempio \(2\)](#)

[Esempio \(3\)](#)

[Esempio \(4\)](#)

[Nuove eccezioni](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

- È possibile usare tipi di eccezioni già presenti nelle Java API, oppure crearne di propri in questo modo:

```
class MiaEccezione extends Exception { }
```

oppure estendendo una qualunque sottoclasse di `Exception`.

- Da questo momento in poi si può lanciare un oggetto del tipo (checked) `MiaEccezione`.
- Pertanto, il codice seguente non compila:

```
class TestEx {  
    void f() {  
        throw new MiaEccezione();  
    }  
}
```



[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

Regole di overriding

Overriding

[Esercizi](#)

[Questionario](#)

Regole di overriding



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

Overriding

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente **le regole complete per l'overriding di metodi:**



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

Overriding

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

Overriding

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

Overriding

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare static uno solo dei metodi



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

Overriding

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare static uno solo dei metodi
- Il metodo nella superclasse non può essere marcato final



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

Overriding

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare static uno solo dei metodi
- Il metodo nella superclasse non può essere marcato final
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Overriding](#)

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare static uno solo dei metodi
- Il metodo nella superclasse non può essere marcato final
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse
- Se il metodo nella sottoclasse dichiara di lanciare un tipo di eccezione *checked*, tale tipo deve essere un sottotipo di una delle eccezioni dichiarate dal metodo della superclasse



Overriding

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Overriding](#)

[Esercizi](#)

[Questionario](#)

Posto che sono sovrapponibili solo i metodi visibili della superclasse, ecco finalmente le **regole complete per l'overriding di metodi**:

- I due metodi devono avere identica segnatura (nome del metodo e tipo dei parametri formali)
- Il tipo di ritorno nella sottoclasse può essere uguale o un sottotipo del tipo di ritorno originario
- Non si può marcare static uno solo dei metodi
- Il metodo nella superclasse non può essere marcato final
- Il metodo nella sottoclasse deve avere visibilità non inferiore a quello della superclasse
- Se il metodo nella sottoclasse dichiara di lanciare un tipo di eccezione *checked*, tale tipo deve essere un sottotipo di una delle eccezioni dichiarate dal metodo della superclasse
Cioè, le **EVENTUALI** eccezioni *checked* dichiarate dal metodo nella sottoclasse, **DEVONO** essere tipi posti al di sotto nella gerarchia delle eccezioni dichiarate dal metodo nella superclasse.



Esercizi

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Esercizi](#)

[Questionario](#)

Sono proposti in allegato due esercizi:

1. In questo esercizio si dovranno usare i blocchi try-catch per gestire una semplice `RuntimeException`.
2. In questo esercizio si dovrà creare il nuovo tipo `OverdraftException` di oggetti lanciabili dal metodo `preleva` nella classe `Conto`.



[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Questionario



D 1

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccB?

- A.** 1
- B.** 2
- C.** 3
- D.** 4
- E.** 5
- F.** 6



D 1

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccB?

- A.** 1
- B.** 2
- C.** 3
- D.** 4
- E.** 5
- F.** 6

B. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



D 2

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6



D 2

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //     +--- EccA  
5.     //             +--- EccB  
6.     //                 +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 non venga lanciata alcuna eccezione?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

A. E. F. – Viene completato il blocco try, eseguito il blocco finally, infine l'esecuzione continua normalmente.



D 3

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccC?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6



D 3

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccC?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

C. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



D 4

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccA?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6



D 4

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo EccA?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

D. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



D 5

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Exception?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6



D 5

Eccezioni: il meccanismo

Catturare le eccezioni

Eccezioni: i dettagli

Regole di overriding

Esercizi

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Exception?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

D. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



D 6

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //          +--- EccB  
6.     //          +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `RuntimeException`?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6



D 6

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

D 6

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)

[D 11](#)

[D 12](#)

[D 13](#)

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //     +--- EccA  
5.     //             +--- EccB  
6.     //                 +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo `RuntimeException`?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

D. E. F. – Viene catturata l'eccezione, eseguito il blocco `finally`, infine l'esecuzione continua normalmente.



D 7

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Error?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6



D 7

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

D 11

D 12

D 13

Dato il codice seguente:

```
1. try {  
2.     // codice rischioso; hp:  
3.     // Exception  
4.     //      +--- EccA  
5.     //              +--- EccB  
6.     //                  +--- EccC  
7.     System.out.print(1);  
8. }  
9. catch (EccB e) {  
10.    System.out.println(2);  
11. }  
12. catch (EccC e) {  
13.    System.out.println(3);  
14. }  
15. catch (Exception e) {  
16.    System.out.println(4);  
17. }  
18. finally {  
19.    System.out.println(5);  
20. }  
21. System.out.println(6);
```

Quali righe saranno presenti nell'output, nel caso in cui, alla riga 2 venga lanciata una eccezione di tipo Error?

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5
- F. 6

E. – Non viene catturata l'eccezione, viene eseguito il blocco finally, l'eccezione propagata.



D 8

```
public class M {  
    public static void  
        main(String[] args) {  
            int k=0;  
            try {  
                int i=5/k;  
            }  
            catch (ArithmetricException e) {  
                System.out.print(1);  
            }  
            catch (RuntimeException e) {  
                System.out.print(2);  
                return;  
            }  
            catch (Exception e) {  
                System.out.print(3);  
            }  
            finally {  
                System.out.print(4);  
            }  
            System.out.print(5);  
        }  
}
```

Qual è l'output del programma precedente?

- A. 5
 - B. 14
 - C. 124
 - D. 145
 - E. 1245
 - F. 35
-



D 8

```
public class M {  
    public static void  
        main(String[] args) {  
            int k=0;  
            try {  
                int i=5/k;  
            }  
            catch (ArithmetricException e) {  
                System.out.print(1);  
            }  
            catch (RuntimeException e) {  
                System.out.print(2);  
                return;  
            }  
            catch (Exception e) {  
                System.out.print(3);  
            }  
            finally {  
                System.out.print(4);  
            }  
            System.out.print(5);  
        }  
}
```

Qual è l'output del programma precedente?

- A. 5
- B. 14
- C. 124
- D. 145
- E. 1245
- F. 35

D.



D 9

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

D 9

[D 10](#)

[D 11](#)

[D 12](#)

[D 13](#)

```
public class Eccezioni {  
    public static void main(String[] args) {  
        try {  
            if (args.length == 0) return;  
            System.out.println(args[0]);  
        }  
        finally {  
            System.out.println("Fine");  
        }  
    }  
}
```

Quali affermazioni riguardanti il precedente programma sono vere?

- A. Se eseguito senza argomenti, il programma non produce output.
- B. Se eseguito senza argomenti, il programma stampa Fine.
- C. Il programma lancia un `ArrayIndexOutOfBoundsException`.
- D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.
- E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da Fine.



D 9

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

D 9

[D 10](#)

[D 11](#)

[D 12](#)

[D 13](#)

```
public class Eccezioni {  
    public static void main(String[] args) {  
        try {  
            if (args.length == 0) return;  
            System.out.println(args[0]);  
        }  
        finally {  
            System.out.println("Fine");  
        }  
    }  
}
```

Quali affermazioni riguardanti il precedente programma sono vere?

- A. Se eseguito senza argomenti, il programma non produce output.
- B. Se eseguito senza argomenti, il programma stampa Fine.
- C. Il programma lancia un `ArrayIndexOutOfBoundsException`.
- D. Se eseguito con un argomento, il programma stampa solo l'argomento dato.
- E. Se eseguito con un argomento, il programma stampa l'argomento dato seguito da Fine.

B. E.



D 10

Qual è l'output del seguente programma?

```
public class MyClass {  
    public static void main(String[] args) {  
        RuntimeException re = null;  
        throw re;  
    }  
}
```

- A. Il codice non viene compilato, poiché il `main` non dichiara che lancia una `RuntimeException`.
- B. Il codice non viene compilato, poiché non può rilanciare `re`.
- C. Il programma viene compilato e lancia `java.lang.RuntimeException` in esecuzione.
- D. Il programma viene compilato e lancia `java.lang.NullPointerException` in esecuzione.
- E. Il programma viene compilato, eseguito e termina senza produrre alcun output.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

D 10

[D 11](#)

[D 12](#)

[D 13](#)



D 10

Qual è l'output del seguente programma?

```
public class MyClass {  
    public static void main(String[] args) {  
        RuntimeException re = null;  
        throw re;  
    }  
}
```

- A. Il codice non viene compilato, poiché il `main` non dichiara che lancia una `RuntimeException`.
 - B. Il codice non viene compilato, poiché non può rilanciare `re`.
 - C. Il programma viene compilato e lancia `java.lang.RuntimeException` in esecuzione.
 - D. Il programma viene compilato e lancia `java.lang.NullPointerException` in esecuzione.
 - E. Il programma viene compilato, eseguito e termina senza produrre alcun output.
-
- D.



D 11

Quali di queste affermazioni sono vere?

- A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.
- B. Un metodo sovrapposto in una sottoclassificazione deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.
- C. Il `main` può dichiarare che lancia eccezioni checked.
- D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclassificazione di quel tipo.
- E. Il blocco `finally` è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco `try`.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)

D 11

[D 12](#)

[D 13](#)



D 11

Quali di queste affermazioni sono vere?

- A. Se una eccezione non è catturata in un metodo, il metodo termina e viene ripresa la successiva normale esecuzione.
- B. Un metodo sovrapposto in una sottoclassificazione deve dichiarare che lancia lo stesso tipo di eccezione del metodo che sovrappone.
- C. Il `main` può dichiarare che lancia eccezioni checked.
- D. Un metodo che dichiara di lanciare una certo tipo di eccezione, può lanciare una istanza di una qualunque sottoclassificazione di quel tipo.
- E. Il blocco `finally` è eseguito se e solo se viene lanciata una eccezione all'interno del corrispondente blocco `try`.

C. D.

[Eccezioni: il meccanismo](#)

[Catturare le eccezioni](#)

[Eccezioni: i dettagli](#)

[Regole di overriding](#)

[Esercizi](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)

D 11

[D 12](#)

[D 13](#)



D 12

```
public class MiaClasse {  
    public static void main  
        (String[] args) {  
        try {  
            f();  
        }  
        catch (MiaEcc e) {  
            System.out.print(1);  
            throw new RuntimeException();  
        }  
        catch (RuntimeException e) {  
            System.out.print(2);  
            return;  
        }  
        catch (Exception e) {  
            System.out.print(3);  
        }  
        finally {  
            System.out.print(4);  
        }  
        System.out.print(5);  
    }  
}
```

```
// MiaEcc è  
// sottoclasse di Exception  
static void f() throws MiaEcc {  
    throw new MiaEcc();  
}  
}
```

Qual è l'output del programma precedente?

- A. 5
 - B. 14
 - C. 124
 - D. 145
 - E. 1245
 - F. 35
-



D 12

```
public class MiaClasse {  
    public static void main  
        (String[] args) {  
        try {  
            f();  
        }  
        catch (MiaEcc e) {  
            System.out.print(1);  
            throw new RuntimeException();  
        }  
        catch (RuntimeException e) {  
            System.out.print(2);  
            return;  
        }  
        catch (Exception e) {  
            System.out.print(3);  
        }  
        finally {  
            System.out.print(4);  
        }  
        System.out.print(5);  
    }  
}
```

```
// MiaEcc è  
// sottoclasse di Exception  
static void f() throws MiaEcc {  
    throw new MiaEcc();  
}  
}
```

Qual è l'output del programma precedente?

- A. 5
- B. 14
- C. 124
- D. 145
- E. 1245
- F. 35

B.



D 13

```
public class MiaClasse {  
    public static void main  
        (String[] args)  
        throws MiaEcc {  
        try {  
            f();  
            System.out.print(1);  
        }  
        finally {  
            System.out.print(2);  
        }  
        System.out.print(3);  
    }  
  
    // MiaEcc e'  
    // sottoclasse di Exception  
    static void f() throws MiaEcc {  
        throw new MiaEcc();  
    }  
}
```

Qual è l'output del programma precedente?

- A. 2 e lancia `MiaEcc`.
 - B. 12
 - C. 123
 - D. 23
 - E. 32
 - F. 13
-



D 13

```
public class MiaClasse {  
    public static void main  
        (String[] args)  
        throws MiaEcc {  
        try {  
            f();  
            System.out.print(1);  
        }  
        finally {  
            System.out.print(2);  
        }  
        System.out.print(3);  
    }  
  
    // MiaEcc e'  
    // sottoclasse di Exception  
    static void f() throws MiaEcc {  
        throw new MiaEcc();  
    }  
}
```

Qual è l'output del programma precedente?

- A. 2 e lancia *MiaEcc*.
 - B. 12
 - C. 123
 - D. 23
 - E. 32
 - F. 13
-

A.

Addendum a lezione 3

Implementazione efficiente dell'ambiente non
locale con scoping statico

Piero Bonatti

Annidamento (nesting)

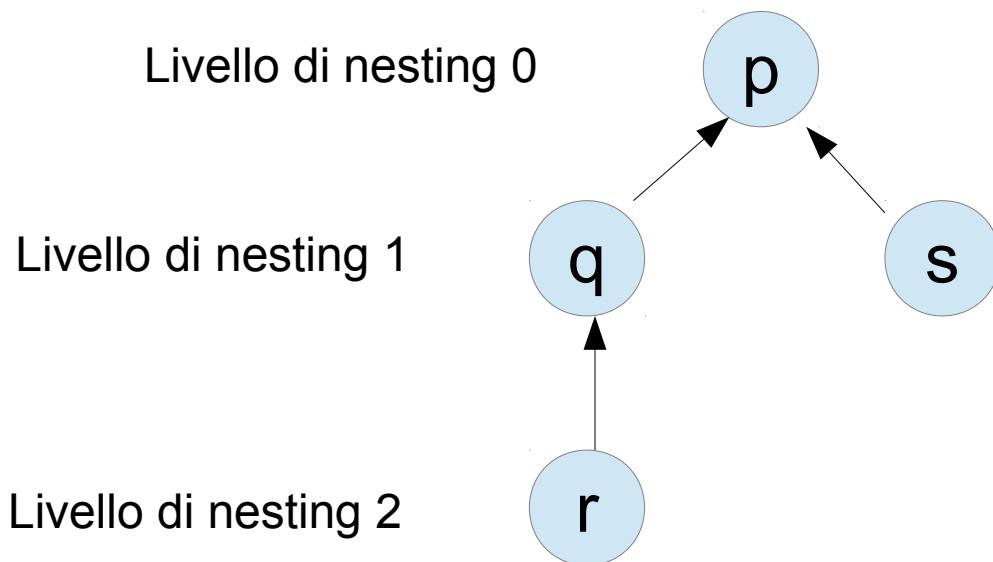
```
program p;  
var a,b,c int;  
  
procedure q;  
var a,c int;  
  
procedure r;  
var a int;  
...  
...  
  
procedure s;  
var b int;  
...  
...
```

- Una procedura q è **annidata** in un blocco b se è definita dentro b, ad esempio:
 - q e s sono annidate in p
 - r è annidata in q
- Il **livello di nesting di una procedura** è il numero di blocchi che la contengono
 - il livello di nesting di q e s è 1
 - quello di r è 2
- L'**ambiente statico non locale** di una procedura è dato dall'ambiente delle procedure in cui è innestata

Annidamento (nesting)

```
program p;  
var a,b,c int;  
  
procedure q;  
var a,c int;  
  
procedure r;  
var a int;  
...  
...  
  
procedure s;  
var b int;  
...  
...
```

- L'annidamento si può rappresentare come un albero
- I livelli dell'albero di nesting corrispondono ai livelli di nesting



Le frecce indicano l'ambiente non locale

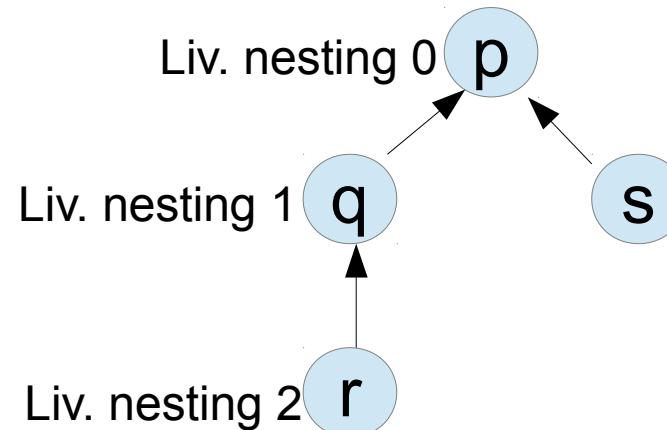
Rappresentazione variabili non locali

```
program p;  
var a,b,c int;
```

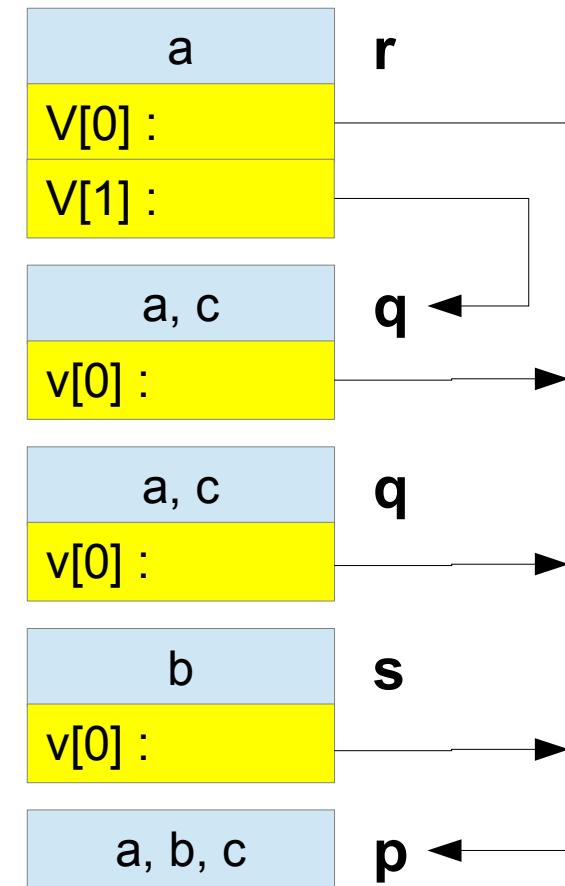
```
procedure q;  
var a,c int;
```

```
procedure r;  
var a int;  
...  
...
```

```
procedure s;  
var b int;  
...  
...
```



Stack di attivazione



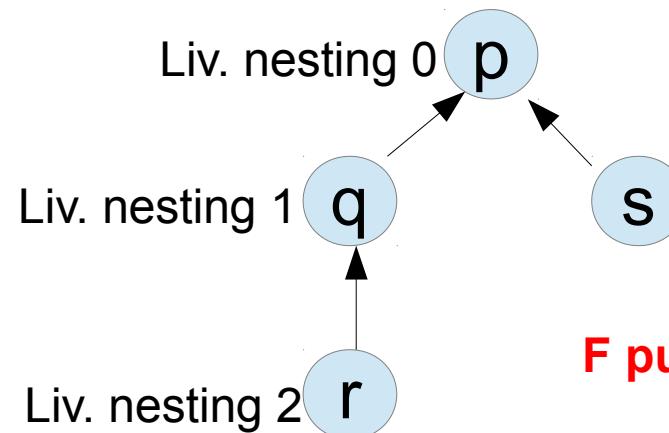
Rappresentazione variabili non locali:
(livello, offset)

Nella procedura r:
b è rappresentata da (0,2)
c è rappresentata da (1,2)

env(x) = v[livello] + offset

Visibilità delle procedure

```
program p;  
var a,b,c int;  
  
procedure q;  
var a,c int;  
  
procedure r;  
var a int;  
...  
...  
  
procedure s;  
var b int;  
...  
...
```



(Ambito statico)

F può chiamare G se:

- G è definita in F
- G è definita in uno dei blocchi che contiene F

quindi:

F e G hanno sempre una parte di ambiente non locale in comune

Mantenimento del vettore degli ambienti non locali

```
program p;  
var a,b,c int;
```

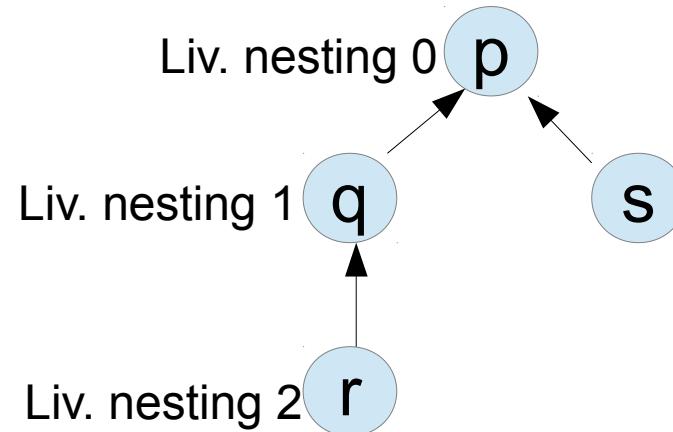
```
procedure q;  
var a,c int;
```

```
procedure r;  
var a int;  
...
```

```
...
```

```
procedure s;  
var b int;  
...
```

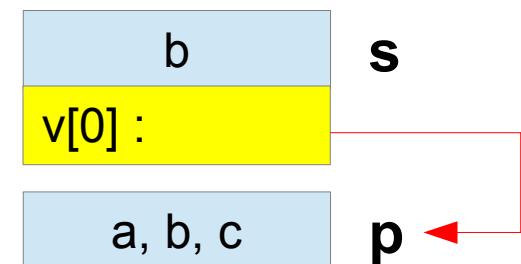
```
...
```



Chiamata a procedure definite localmente:

Esempio 1: p chiama s:

- Aumenta liv. di nesting
- Aggiungere elem. a v[]



Mantenimento del vettore degli ambienti non locali

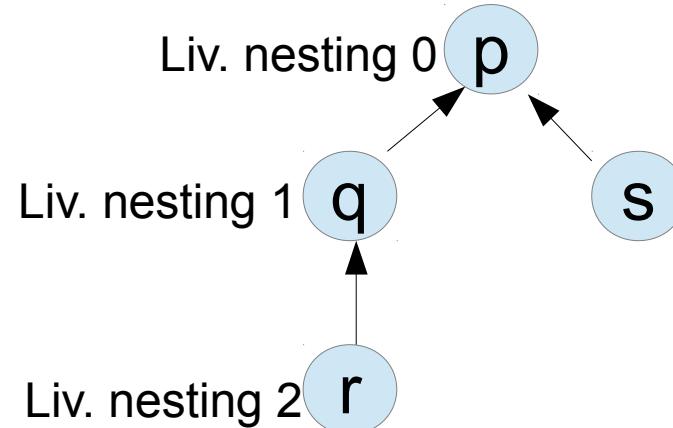
```
program p;  
var a,b,c int;
```

```
procedure q;  
var a,c int;
```

```
procedure r;  
var a int;  
...
```

```
...
```

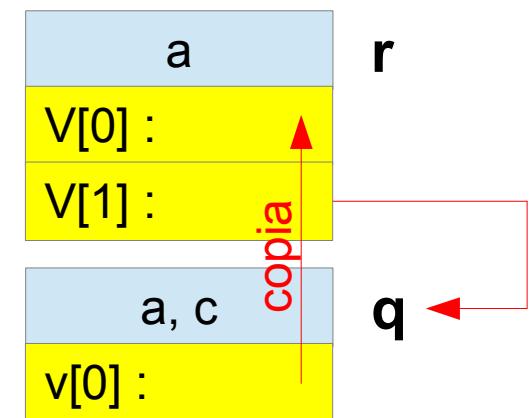
```
procedure s;  
var b int;  
...
```



Chiamata a procedure definite localmente:

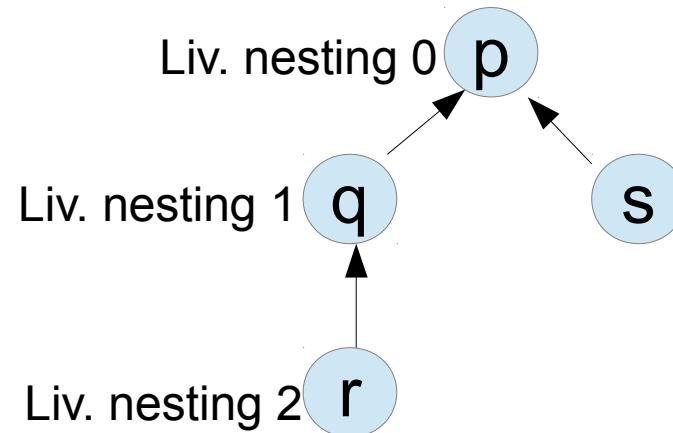
Esempio 2: q chiama r:

- Aumenta liv. di nesting
- Aggiungere elem. a v[]
- Copiare gli altri elementi



Mantenimento del vettore degli ambienti non locali

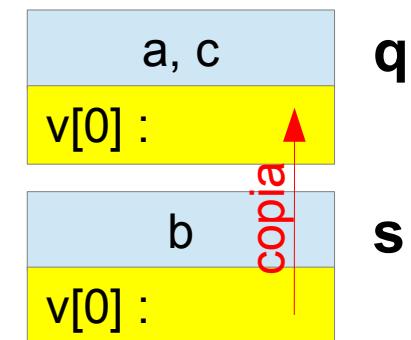
```
program p;  
var a,b,c int;  
  
procedure q;  
var a,c int;  
  
procedure r;  
var a int;  
...  
...  
  
procedure s;  
var b int;  
...  
...
```



Chiamata a procedure definite **esternamente**:

Esempio 1: s chiama q:

- stesso liv. di nesting
- Copiare gli elementi di v[]



Mantenimento del vettore degli ambienti non locali

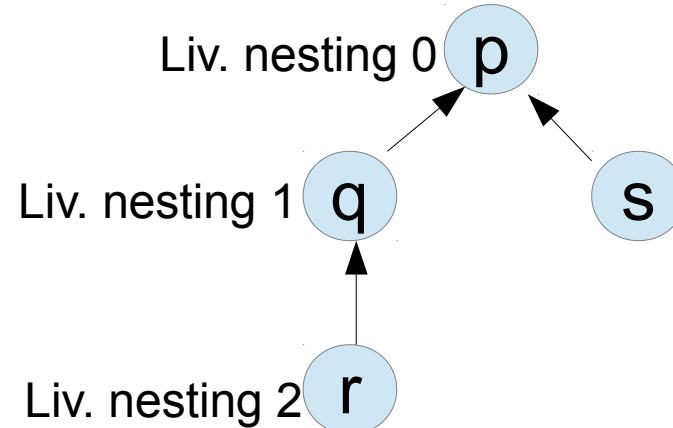
```
program p;  
var a,b,c int;
```

```
procedure q;  
var a,c int;
```

```
procedure r;  
var a int;  
...
```

```
...
```

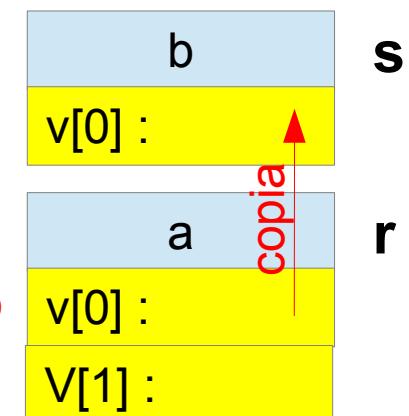
```
procedure s;  
var b int;  
...
```



Chiamata a procedure definite esternamente:

Esempio 2: r chiama s:

- il liv. di nesting **decresce**
- Copiare solo gli elementi di livello *minore o uguale* a quello di s
- Struttura ad albero + scoping statico garantiscono sempre di trovare l'ambiente non locale della procedura chiamata nel record del chiamante



Proprietà di questa implementazione

- Accesso alle variabili in tempo costante
 - Accesso a vettore + somma del puntatore ivi contenuto e dell'offset
 - Indipendente dai livelli di nesting
 - Calcolo supportato direttamente dalle istruzioni macchina
- Creazione record di attivazione lineare nel livello di nesting (copia degli elementi di $v[]$)
 - Indipendente dall'esecuzione
 - Dipende solo dal testo del sorgente
- Tempo **costante**
- L'implementazione naïve richiederebbe a run time di percorrere le liste di puntatori all'ambiente non locale
 - Accesso alle variabili in tempo **lineare** nel livello di nesting
 - Creazione dei record di attivazione in tempo **costante** nel livello di nesting

Introduzione ai generics

**Polimorfismo parametrico
vs
polimorfismo per inclusione**

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - Push(element)
 - Pop()
 - Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - Implementazione con array

Una soluzione

- Sfruttare Object

```
public class ObjectStack {  
  
    private Object[] stack;  
    private int top = -1;  
  
    public ObjectStack( int capacity ) { stack = new Object [capacity]; }  
  
    public void push( Object el ) { stack[++top] = el; }  
  
    public Object pop() { return stack[top--]; }  
}
```

Polimorfismo per inclusione,
Polimorfismo di sottotipo

Una soluzione

■ Esempio di uso #1

```
ObjectStack so = new ObjectStack( 10 );  
  
so.push("a");  
so.push("b");  
so.push("c");  
String msg = "";  
for (int i = 0; i<3; i++ ) {  
    String x = (String) so.pop();  
    msg += x;  
}  
System.out.println(msg);
```

Se non faccio il downcast,
non posso assegnare a String

cba

output

Una soluzione

- Esempio di uso #2

```
ObjectStack so = new ObjectStack( 10 );

so.push("a");
so.push("b");
so.push( Integer.valueOf(42) );
String msg = "";
for (int i = 0; i<3; i++ ) {
    String x = (String) so.pop();
    msg += x;
}
System.out.println(msg);
```

Se mi distraggo compila ancora ma...

output

java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String

Una soluzione migliore: *generics*

- Ovvero una **classe parametrica**

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int capacity ) { stack = new ElemType[ capacity ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Parametro formale di tipo

Ci piacerebbe...
ma non si può fare!

Una soluzione migliore: *generics*

- Esempio di uso #1

```
GenericStack< String > gs = new GenericStack< String >( 10 );  
  
gs.push("a");  
gs.push("b");  
gs.push("c");  
String msg = "";  
for (int i = 0; i<3; i++ ) {  
    String x = gs.pop();  
    msg += x;  
}  
System.out.println(msg);
```

Parametro *attuale* di tipo

Non serve downcast:
pop() restituisce **String**

cba

output

Una soluzione migliore: *generics*

- Esempio di uso #2

```
GenericStack< String > gs = new GenericStack< String >( 10 );
```

```
gs.push("a");  
gs.push("b");  
gs.push( Integer.valueOf(42) );  
...  
...
```

push(**String**)

1. ERROR in Test.java (at line 8)

```
gs.push( Integer.valueOf(42) );  
^^^^^
```

output del compilatore!

The method push(String) in the type GenericStack<String> is not applicable
for the arguments (Integer)

Confronto

- Pro del polimorfismo parametrico:
 - Non richiede controlli di tipo a run time
 - Anticipa scoperta errori di tipo a tempo di compilazione
- Pro del polimorfismo per inclusione
 - Permette strutture dati eterogenee
 - Ad esempio, Stack di elementi di tipo diverso

Prendere il meglio

- In molti casi
 - Servono collezioni di elementi eterogenei
 - Ma vogliamo usarli allo stesso modo
- Esempio:
 - Una scena è una *lista* di forme eterogenee (rettangoli, ellissi, linee, ecc.)
 - Vogliamo usarla per implementare *refresh*, che deve solo inviare un messaggio *draw()* a tutte le forme della lista
- Quindi
 - Identificare le modalità d'uso degli elementi
 - Scegliere una superclasse comune (o fattorizzarle in una *interfaccia*)
 - Usare la superclasse/interfaccia come argomento del template

Note sull'implementazione

- In Java – dopo la compilazione - diventa comunque uno stack di Object
 - I parametri di tipo vengono usati per il type checking e poi **cancellati (erasure)**
 - Per questo un parametro attuale di tipo non può essere primitivo come int o float, ma dobbiamo usare i wrapper
 - Per questo non si può istanziare un array di un tipo parametrico

Note sull'implementazione

- In C++, ogni uso di un template fa compilare una nuova classe
 - Sorta di macro
 - Il compilatore inserisce nel programma la definizione di classe specializzata e la ricompila per ogni parametro attuale
 - Più espressivo ma anche più “costoso”
 - Si può usare un parametro di tipo in tutti i modi in cui si potrebbe usare un tipo vero

Array parametrici in Java

- Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int capacity ) { stack = (ElemType[]) new Object[ capacity ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Siamo costretti a un downcast

- Otteniamo comunque un warning a tempo di compilazione

Array parametrici in Java

- Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    @SuppressWarnings("unchecked")  
    public GenericStack( int capacity ) { stack = (ElemType[]) new Object[ capacity ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Annotazione

- Ora niente warning
- Meglio ancora: usare ArrayList<ElemType>

2. Esempio di utilizzo delle interfacce

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - Push(element)
 - Pop()
 - Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - **Non “forzare” una specifica implementazione**
 - **Se ne occupa un altro team**
 - **Serve un modo per far lavorare i due team indipendentemente ma garantendo l'integrazione dei risultati**

Soluzione: interfacce

- Definisco un'interfaccia con i metodi richiesti

```
public interface Stack<ElemType> {  
    public void push( ElemType el );  
    public ElemType pop();  
}
```

Implementare l'interfaccia

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int capacity ) { ... }  
  
    public void push( ElemType el ) { stack [++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Utilizzare l'interfaccia

■ Esempio

```
Stack< Integer > si = new GenericStack< Integer >( 10 );  
  
si.push(1);  
si.push(2);  
si.push(3);  
for( int i = 0; i<3; i++ ) {  
    System.out.println( si.pop().intValue() );  
}
```

Posso cambiare implementazione
semplicemente cambiando qs tipo.
Il resto del codice resta uguale.

3
2
1

output

Confronto con classi astratte

- Se avessi definito Stack come una classe astratta...
 - ...GenericStack avrebbe **dovuto estendere** Stack
 - In Java, una classe può estendere una sola altra classe
 - Restringe l'insieme di classi che si possono utilizzare
- Vantaggi delle interfacce
 - Non impone lo stesso obbligo
 - Le classi che implementano l'interfaccia Stack possono essere ovunque nella gerarchia delle classi
- Vantaggi delle classi astratte
 - Posso contenere dati (aka *stato*)

3. Esempio di Stack con eccezioni

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - Push(element)
 - Pop()
 - Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - Non “forzare” una specifica implementazione
 - **Gestire pulitamente gli errori specifici**
 - **Push su stack pieno**
 - **Pop su stack vuoto**

Soluzione: eccezioni

- Cosa succede attualmente con troppi push o troppi pop?

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int capacity ) { ... }  
  
    public void push( ElemType el ) { stack [++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Soluzione: eccezioni

- Introduciamo due eccezioni custom, non verificate

```
public class EmptyStackException extends RuntimeException {  
    public EmptyStackException() {}  
    public EmptyStackException(String msg) { super(msg); }  
}  
  
public class FullStackException extends RuntimeException {  
    public FullStackException() {}  
    public FullStackException(String msg) { super(msg); }  
}
```

Implementare l'interfaccia

■ Lanciamo le eccezioni

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int capacity ) { ... }  
  
    public void push( ElemType el ) {  
        if ( top<stack.length-1 ) stack[++top] = el;  
        else throw new FullStackException()  
    }  
  
    public ElemType pop() {  
        if( top >= 0 ) return stack[ top-- ];  
        else throw new EmptyStackException();  
    }  
}
```

Utilizzo

■ Esempio di svuotamento stack

```
Stack< Integer > si = new GenericStack< Integer >();
```

```
si.push(1);
si.push(2);
si.push(3);
try {
    while( true ) {
        System.out.println( si.pop().intValue() );
    }
} catch( EmptyStackException e ) {
    System.out.println( "fine" );
}
```

Non serve downcast:
pop() restituisce Integer

3
2
1
fine

output

Linguaggi di Programmazione I – Lezione 1

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

10 marzo 2022



Panoramica della lezione

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Paradigmi computazionali



[Introduzione al corso](#)

[Obiettivi specifici](#)

[Obiettivi generali](#)

[Scheduling](#)

[Altre informazioni](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

Introduzione al corso



Obiettivi specifici

[Introduzione al corso](#)

Obiettivi specifici

[Obiettivi generali](#)

[Scheduling](#)

[Altre informazioni](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

- Mettere lo studente in grado di imparare velocemente un nuovo linguaggio di programmazione
 - ◆ mediante astrazione delle *caratteristiche costituenti* dei linguaggi
 - ◆ cambiano più di rado di quanto vengano introdotti nuovi linguaggi
- Spiegare le differenze tra i vari paradigmi di programmazione - imperativo, ad oggetti, funzionale, logico - e sul loro impatto sullo stile di soluzione dei problemi.
- Mostrare cenni sui criteri di progetto e le tecniche d'implementazione dei linguaggi di programmazione
- Esempi focalizzati su Java, ML, Prolog. Ma anche C, Pascal, e Python.



Obiettivi generali

- Migliorare l'abilità nel risolvere i problemi usando meglio i linguaggi di programmazione.

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di
programmazione)

Macchine astratte

Paradigmi
computazionali



Obiettivi generali

- Migliorare l'abilità nel risolvere i problemi usando meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.

[Introduzione al corso](#)

[Obiettivi specifici](#)

Obiettivi generali

[Scheduling](#)

[Altre informazioni](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Obiettivi generali

- Migliorare l'abilità nel risolvere i problemi usando meglio i linguaggi di programmazione.
- Imparare a scegliere più intelligentemente, in dipendenza del problema, il linguaggio di programmazione.
- Aumentare la capacità di imparare linguaggi di programmazione (che sono TANTI!).

[Introduzione al corso](#)

[Obiettivi specifici](#)

Obiettivi generali

[Scheduling](#)

[Altre informazioni](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Scheduling

[Introduzione al corso](#)

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di
programmazione)

[Macchine astratte](#)

Paradigmi
computazionali

Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.



Scheduling

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di
programmazione)

Macchine astratte

Paradigmi
computazionali

■ Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.

■ Parte seconda (modello object-oriented, Java)

1. Studio dei costrutti più avanzati: qualificatori di classi, metodi e attributi; classi astratte; interfacce; template; classi interne; gestione degli errori: eccezioni.



Scheduling

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di
programmazione)

Macchine astratte

Paradigmi
computazionali

■ Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.

■ Parte seconda (modello object-oriented, Java)

1. Studio dei costrutti più avanzati: qualificatori di classi, metodi e attributi; classi astratte; interfacce; template; classi interne; gestione degli errori: eccezioni.

■ Parte terza (Linguaggi funzionali, ML)

1. ML e costrutti funzionali: funzioni di ordine superiore; polimorfismo; tipi di dato astratti e interfacce; template; gestione degli errori: eccezioni; type inference.



Scheduling

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di programmazione)

Macchine astratte

Paradigmi computazionali

■ Parte prima

1. Storia e concetti di base
2. Primi cenni ai paradigmi dei linguaggi di programmazione.
3. Il modello imperativo.

■ Parte seconda (modello object-oriented, Java)

1. Studio dei costrutti più avanzati: qualificatori di classi, metodi e attributi; classi astratte; interfacce; template; classi interne; gestione degli errori: eccezioni.

■ Parte terza (Linguaggi funzionali, ML)

1. ML e costrutti funzionali: funzioni di ordine superiore; polimorfismo; tipi di dato astratti e interfacce; template; gestione degli errori: eccezioni; type inference.

■ Parte quarta (Linguaggi logici, Prolog)

1. Costrutti fondamentali: termini, predicati e regole.
2. Tecniche di programmazione avanzate: invertibilità e nondeterminismo.



Altre informazioni

■ *Libri di testo:*

- ◆ Dershem - Jipping. *Programming languages: structures and models.*
- ◆ Gabbrielli - Martini. *Linguaggi di programmazione: Principi e paradigmi*
- ◆ Altro materiale fornito dal docente

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di
programmazione)

Macchine astratte

Paradigmi
computazionali



Altre informazioni

Introduzione al corso

Obiettivi specifici

Obiettivi generali

Scheduling

Altre informazioni

Linguaggi (di programmazione)

Macchine astratte

Paradigmi computazionali

■ *Libri di testo:*

- ◆ Dershem - Jipping. *Programming languages: structures and models.*
- ◆ Gabbrielli - Martini. *Linguaggi di programmazione: Principi e paradigmi*
- ◆ Altro materiale fornito dal docente

■ *Materiali:* Sul gruppo Teams e su <http://wpage.unina.it/mfaella>

■ *Ricevimento studenti:* lunedì, 14-16, in presenza o in remoto nel gruppo Teams “Ricevimento Faella”



Introduzione al corso

Linguaggi (di
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali

Linguaggi (di programmazione)



Sono ~ 40?

Introduzione al corso

Linguaggi (di programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

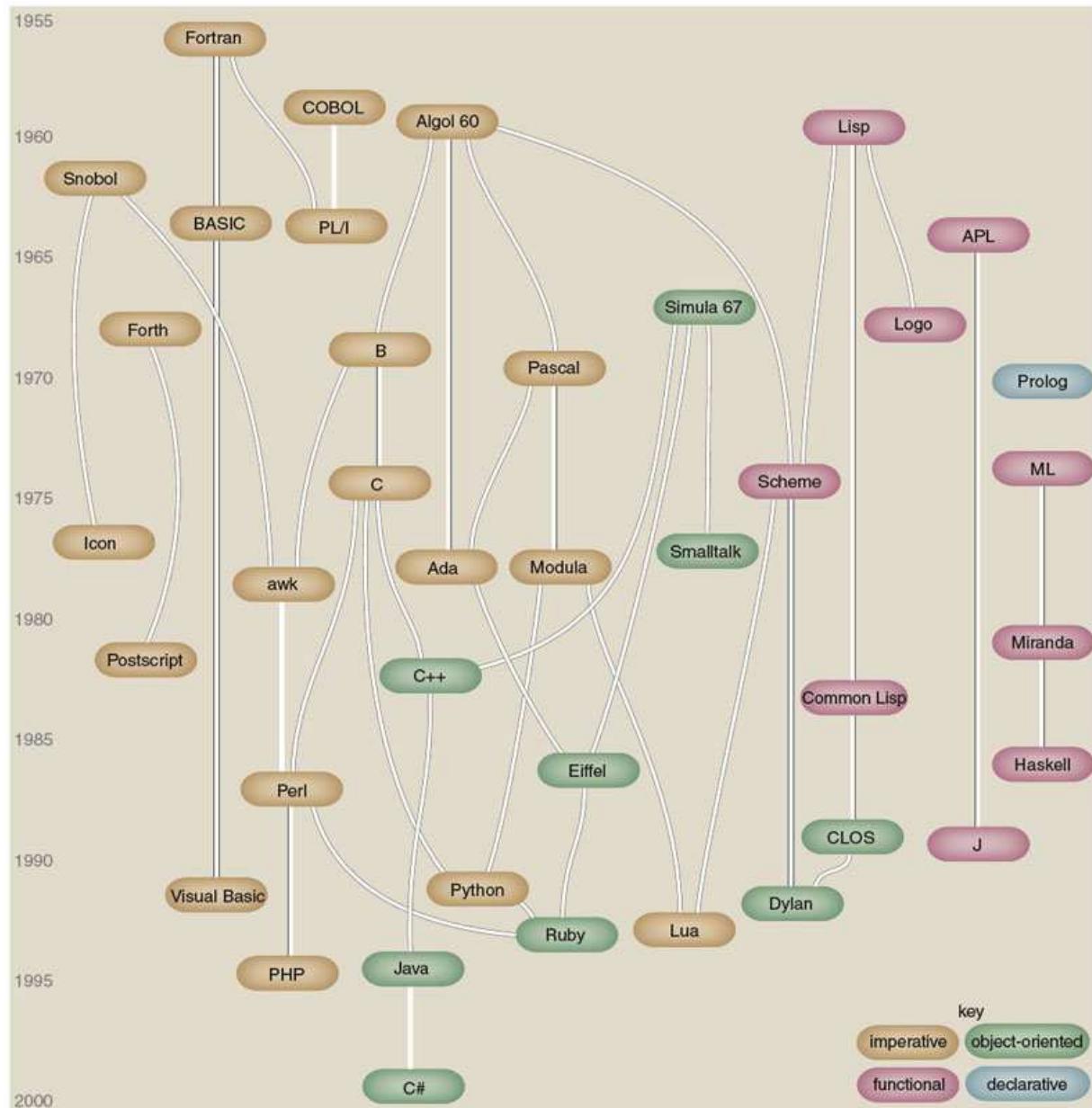
. . . concetti
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali





Sono ~ 80?

Introduzione al corso

Linguaggi (di programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi computazionali

Mother Tongues

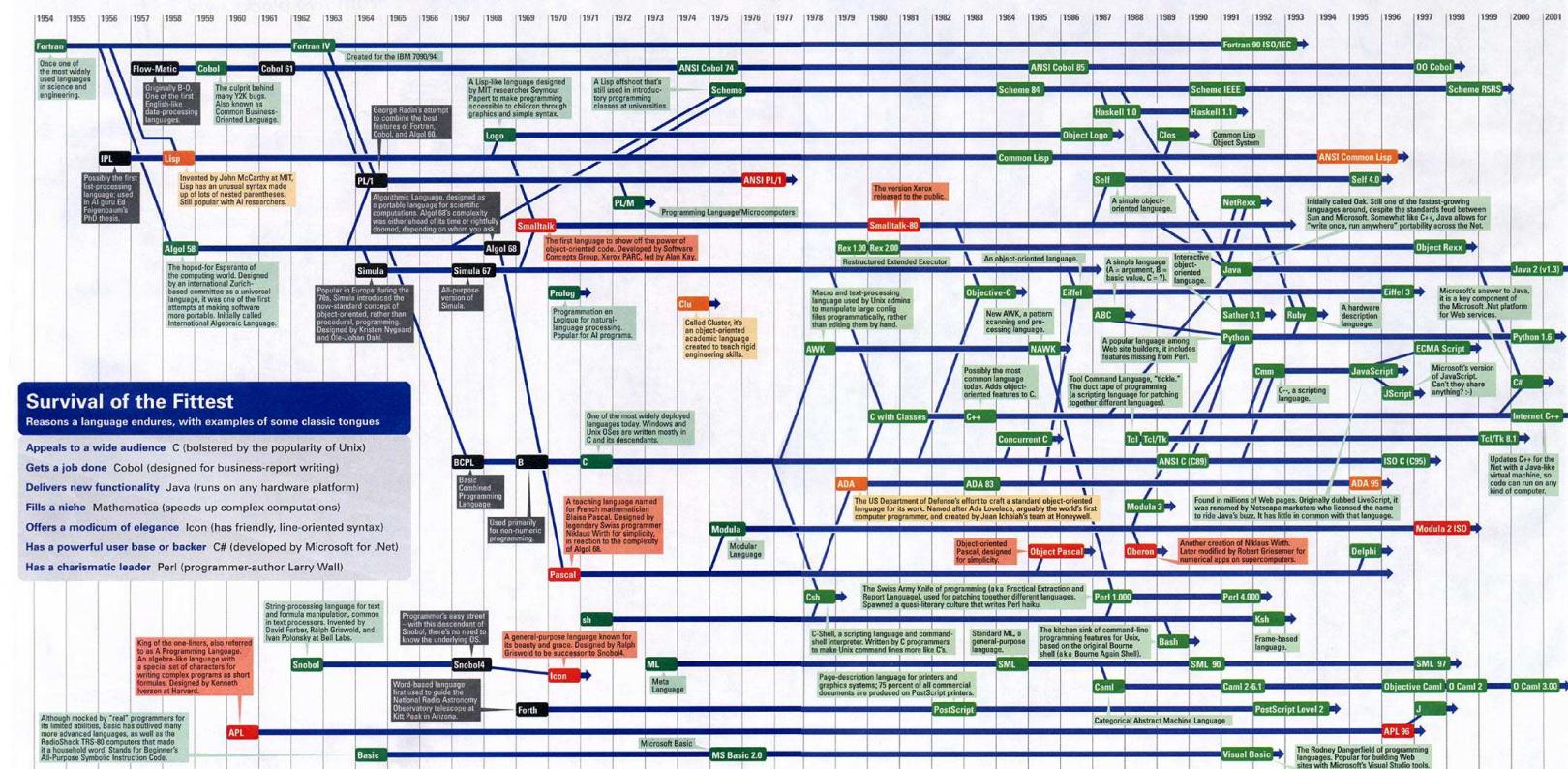
Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C/C++, Visual Basic, Cobol, Java, and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers—electronic lexicographers, if you will—aim to save, or at least document, the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at www.informatik.uni-freiburg.de/Java/misc/lang_list.html—Michael Menduno

| Key | Year Introduced |
|---|--|
| Once one of the most widely used languages in science and engineering. | Active: thousands of users |
| Originally B-O, designed by two Kemeny brothers known as Common Business-Oriented Language. | Protected: taught at universities; compilers available |
| Flow-Matic | Endangered: usage dropping off |
| Created for the IBM 7090/94. | Extinct: no known active users or up-to-date compilers |
| IPL | Lineage continues |





Quanti sono?

Introduzione al corso

Linguaggi (di programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

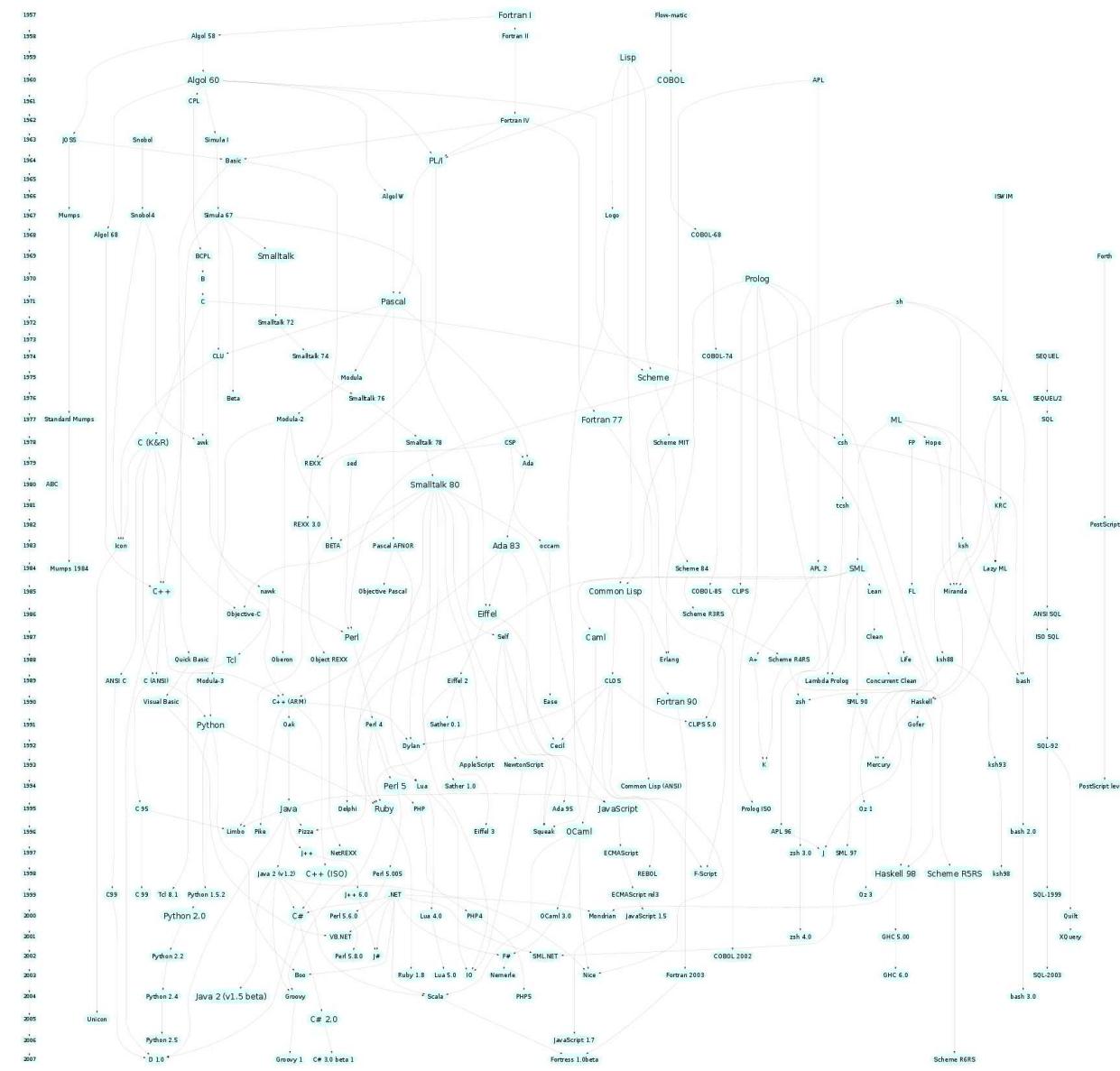
. . . concetti
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali





Quanti sono?

- C'è chi dice addirittura qualche migliaio
 - ◆ Come orientarsi?
 - ◆ Come usarli *bene*?
 - ◆ Come apprendere in fretta quelli nuovi?
- Occorre comprensione astratta delle caratteristiche dei linguaggi per coglierne somiglianze/differenze
- e per comprendere lo scopo di ciascun costrutto (ovvero i principi del *language design*)

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti
introdotti

Terminologia

Linguaggi completi

[Macchine astratte](#)

[Paradigmi
computazionali](#)



Breve storia

1954

FORTRAN (FORmula TRANslation)

Introduzione al corso

Linguaggi (di
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali



Breve storia

Introduzione al corso

Linguaggi (di
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali

| | |
|------|---|
| 1954 | FORTRAN (FORmula TRANslation) |
| 1960 | COBOL (Common Business Oriented Language) ALGOL 60 (Algorithmic Oriented Language) PL/1 (Programming Language 1) Simula 67 ALGOL 68 PASCAL LISP (LISt Processing) APL BASIC |



Breve storia

Introduzione al corso

Linguaggi (di
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti
introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali

| | |
|---------|---|
| 1954 | FORTRAN (FORmula TRANslation) |
| 1960 | COBOL (Common Business Oriented Language) ALGOL 60 (Algorithmic Oriented Language) PL/1 (Programming Language 1) Simula 67 ALGOL 68 PASCAL LISP (LISt Processing) APL BASIC |
| 1970/80 | PROLOG SMALLTALK C MODULA/2 ADA |



Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output.

Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.



Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output.

Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

Cobol: indipendenza dalla macchina e statement “English like”. Orientato ai database. Introduce il record.



Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output.

Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

Cobol: indipendenza dalla macchina e statement “English like”. Orientato ai database. Introduce il record.

Algol60: indipendenza dalla macchina

e definizione mediante grammatica (*Backus-Naur form*), strutture a blocco, supporto generale dell'iterazione e ricorsione.



Storia dei concetti introdotti dai progenitori dell'oggi

Fortran: nato per manipolazione algebrica; introduce: variabili, statement di assegnazione, concetto di tipo, subroutine, iterazione e statement condizionali, go to, formati di input e output.

Gestione solo statica della memoria, no ricorsione, no strutture dinamiche, no tipi definiti da utente.

Cobol: indipendenza dalla macchina e statement “English like”. Orientato ai database. Introduce il record.

Algol60: indipendenza dalla macchina

e definizione mediante grammatica (*Backus-Naur form*), strutture a blocco, supporto generale dell'iterazione e ricorsione.

Lisp: primo vero linguaggio di manipolazione simbolica, paradigma funzionale, non c'è lo statement di assegnazione, e quindi concettualmente non c'è “il valore” ovvero l'idea di cambiare lo stato della memoria. Non c'è differenza concettuale fra funzione e dato: dipende dall'uso. La prima versione era essenzialmente non tipata.



Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (*generate and test*). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.



Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (*generate and test*). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

Simula 67: classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di

dato astratto implementato in Ada e Modula2, e del concetto moderno di classe di Smalltalk e C++.



Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (*generate and test*). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

Simula 67: classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di

dato astratto implementato in Ada e Modula2, e del concetto moderno di classe di Smalltalk e C++.

PL/1: abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; “multitasking”, cioè specificazione di tasks che possono essere eseguiti in concorrenza.



Storia dei concetti introdotti dai progenitori dell'oggi

Prolog: primo (e principale) linguaggio di programmazione logica (paradigma logico). Tra le caratteristiche innovative: invertibilità, programmazione in stile nondeterministico (*generate and test*). Essenzialmente non tipato; estensioni (tipi e altro) mediante metaprogrammazione.

Simula 67: classe come encapsulamento di dati e procedure, istanze delle classi (oggetti): anticipatorio del concetto di tipo di

dato astratto implementato in Ada e Modula2, e del concetto moderno di classe di Smalltalk e C++.

PL/1: abilità ad eseguire procedure specificate quando si verifica una condizione eccezionale; “multitasking”, cioè specificazione di tasks che possono essere eseguiti in concorrenza.

Pascal: programmazione strutturata, tipi di dato definiti da utente, ricchezza di strutture dati. Ma ancora niente encapsulation; si dovrà aspettare Modula.



Terminologia

- **Linguaggio di programmazione:** è un linguaggio che è usato per esprimere (mediante un programma) un algoritmo con il quale un processore può risolvere un problema.

Introduzione al corso

Linguaggi (di
programmazione)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

Linguaggi completi

Macchine astratte

Paradigmi
computazionali



Terminologia

- **Linguaggio di programmazione:** è un linguaggio che è usato per esprimere (mediante un programma) un algoritmo con il quale un processore può risolvere un problema.
- **Programma:** è la codifica di un algoritmo in un linguaggio di programmazione.
- **Processore:** è la macchina (reale o virtuale) che eseguirà l'algoritmo descritto dal programma. Non si deve intendere come un singolo oggetto, ma come una *architettura di elaborazione*.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Terminologia

- **Linguaggio di programmazione:** è un linguaggio che è usato per esprimere (mediante un programma) un algoritmo con il quale un processore può risolvere un problema.
- **Programma:** è la codifica di un algoritmo in un linguaggio di programmazione.
- **Processore:** è la macchina (reale o virtuale) che eseguirà l'algoritmo descritto dal programma. Non si deve intendere come un singolo oggetto, ma come una *architettura di elaborazione*.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono esprimere *qualunque funzione calcolabile*
 - ◆ detti anche *general purpose*
 - ◆ tecnicamente: devono poter simulare *qualunque macchina di Turing*.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

Linguaggi completi

[Macchine astratte](#)

[Paradigmi computazionali](#)



Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono esprimere *qualunque funzione calcolabile*
 - ◆ detti anche *general purpose*
 - ◆ tecnicamente: devono poter simulare qualunque *macchina di Turing*.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono esprimere *qualunque funzione calcolabile*
 - ◆ detti anche *general purpose*
 - ◆ tecnicamente: devono poter simulare qualunque *macchina di Turing*.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile).

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono esprimere *qualunque funzione calcolabile*
 - ◆ detti anche *general purpose*
 - ◆ tecnicamente: devono poter simulare qualunque *macchina di Turing*.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile). È spesso immerso in linguaggi completi.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



Linguaggi completi

- È uso comune intendere come linguaggi di programmazione solo quelli *computazionalmente completi*, cioè solo quelli che possono esprimere *qualunque funzione calcolabile*
 - ◆ detti anche *general purpose*
 - ◆ tecnicamente: devono poter simulare qualunque *macchina di Turing*.
- Sono completi solo quelli che riescono ad esprimere anche programmi di cui non è decidibile la terminazione
- SQL non è un linguaggio completo (perché la terminazione dei programmi è sempre decidibile). È spesso immerso in linguaggi completi.
- HTML non è un linguaggio completo (idem).
- Per mostrare la completezza di un linguaggio: usarlo per simulare arbitrarie macchine di Turing

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Sono ~ 40?

Sono ~ 80?

Quanti sono?

Quanti sono?

Breve storia

Storia dei . . .

. . . concetti

introdotti

Terminologia

[Linguaggi completi](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)



[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

Macchine astratte



Definizione

Dato un linguaggio di programmazione L , una macchina astratta per L (in simboli, M_L) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L .

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Definizione

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

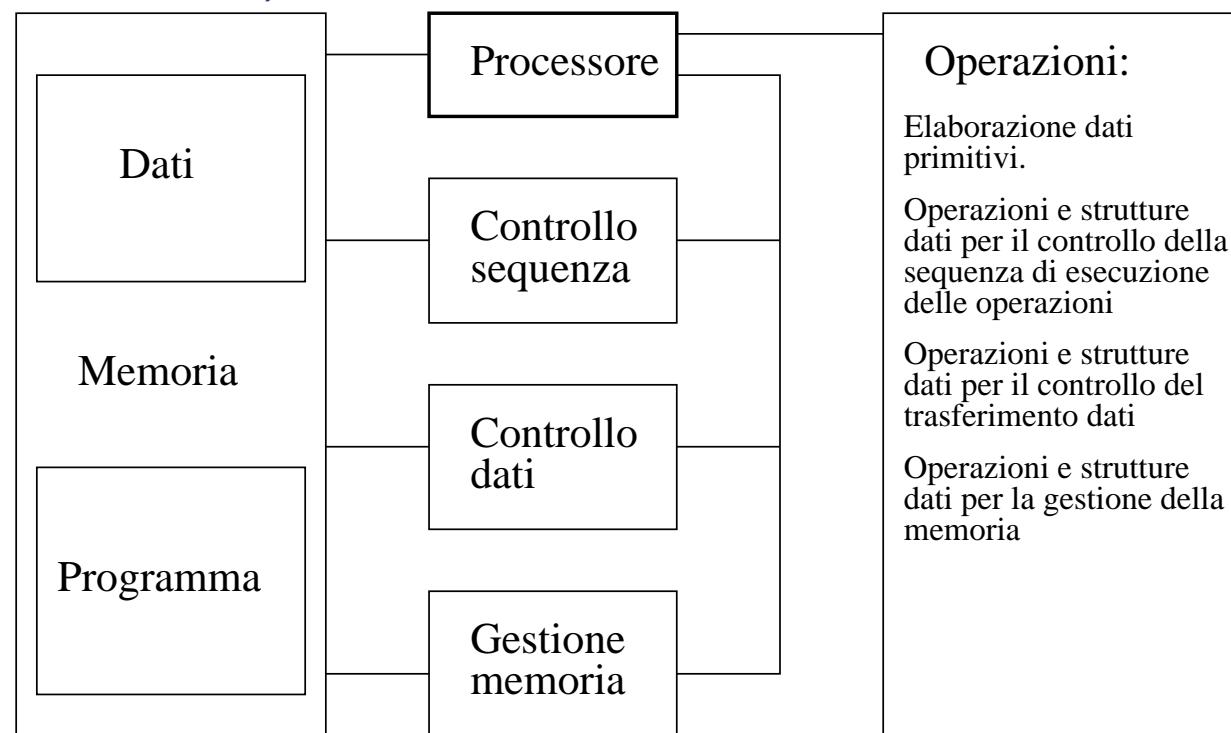
Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

Dato un linguaggio di programmazione L , una macchina astratta per L (in simboli, M_L) è un qualsiasi insieme di strutture dati e algoritmi che permettano di memorizzare ed eseguire programmi scritti in L .

La struttura di una macchina astratta è (essenzialmente memoria e processore):





Esempio

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali

Macchina E–Business (commercio online)

Macchina Web Service

Macchina Web

Macchina linguaggio di programmazione

Macchina intermedia (java bytecode)

Macchina Sistema Operativo

Macchina firmware

Macchina hardware



Processore della macchina astratta

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

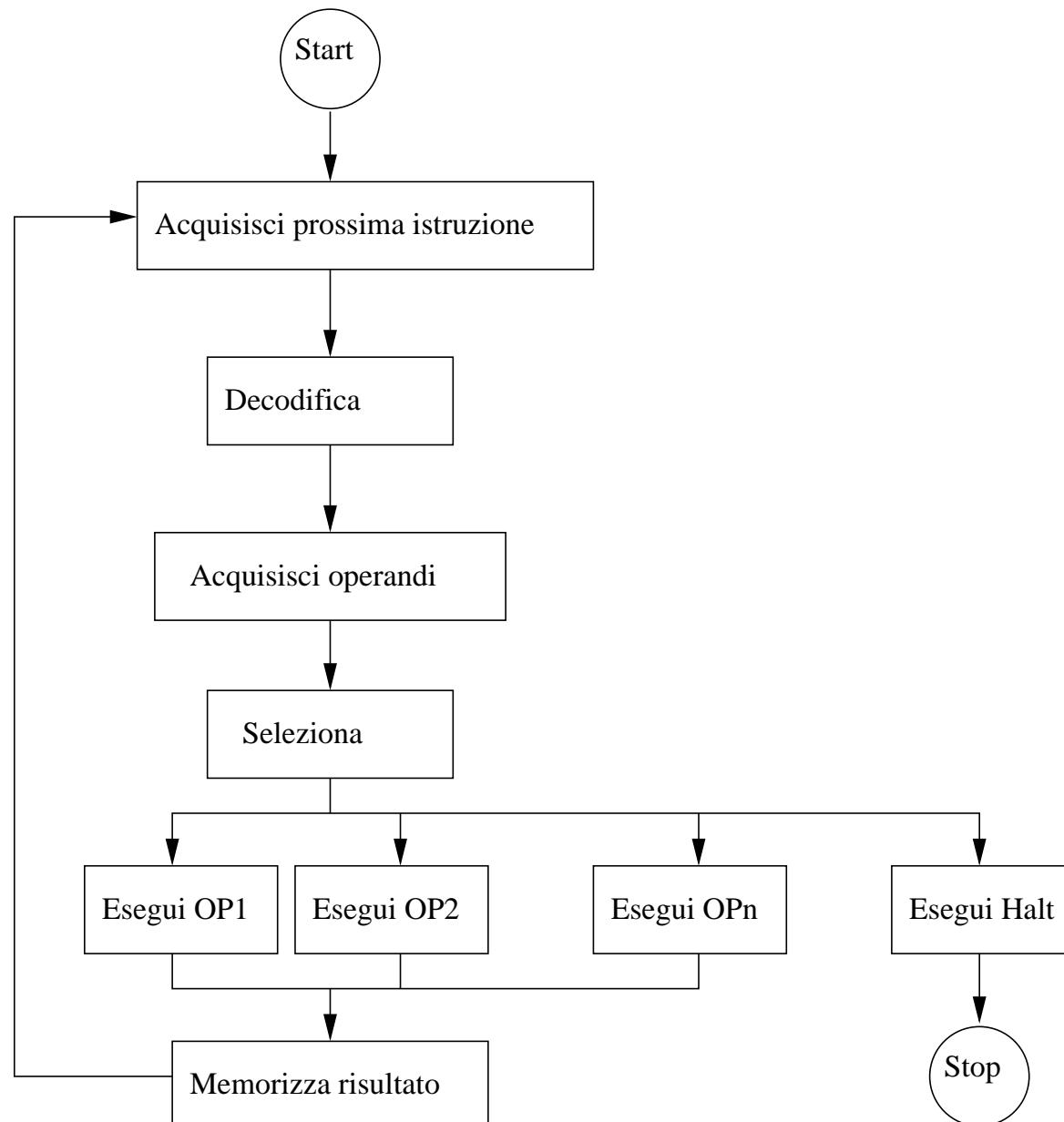
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali





Esempi di macchine astratte

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

| Linguaggio | Macchina astratta |
|------------|-----------------------|
| C | gcc + OS + HW |
| Java | javac + JVM + OS + HW |
| C# | csc + CLR + OS + HW |
| Python | python + OS + HW |



Tecnologie di realizzazione di macchina astratta

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali

Hardware: Si era pensato per Lisp e Prolog



Tecnologie di realizzazione di macchina astratta

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

Hardware: Si era pensato per Lisp e Prolog

Firmware: Soluzione più flessibile e semplice / economicità di
progetto



Tecnologie di realizzazione di macchina astratta

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

Hardware: Si era pensato per Lisp e Prolog

Firmware: Soluzione più flessibile e semplice / economicità di
progetto

Software: Ad es. Java Virtual Machine, o Warren Abstract
Machine (Prolog)



Traduttori Linguaggio → Macchina astratta

- **Interpreti:** traducono ed eseguono un costrutto alla volta.
PRO: debug - fase di sviluppo: interazione più snella.

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori
Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Traduttori Linguaggio → Macchina astratta

- **Interpreti:** traducono ed eseguono un costrutto alla volta.
PRO: debug - fase di sviluppo: interazione più snella.
- **Compilatori:** prima traducono l'intero programma; poi la traduzione può essere eseguita (anche più volte).
PRO: velocità di esecuzione finale - fase di rilascio.
PRO: più controlli e in anticipo

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori
Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Interpretazione pura

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

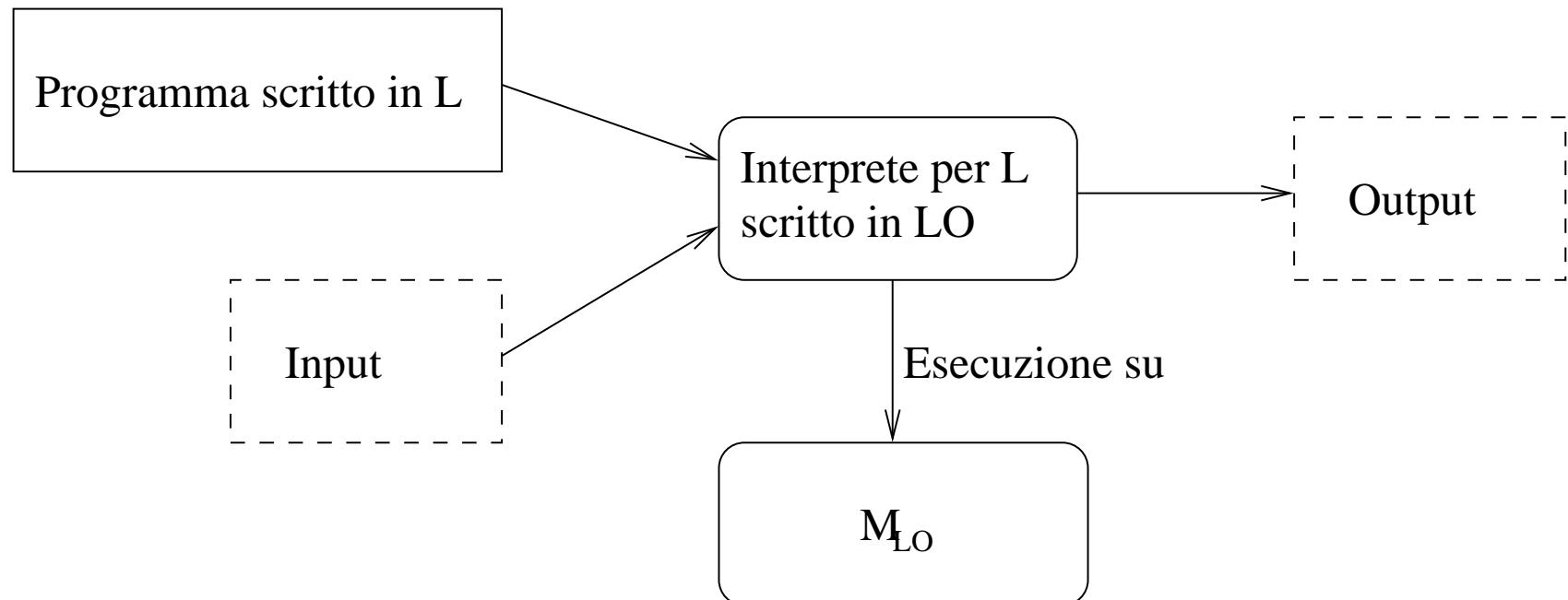
Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



E' il caso dei linguaggi di scripting: bash, awk, Python, etc.



Compilazione pura (caso semplice)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio → Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

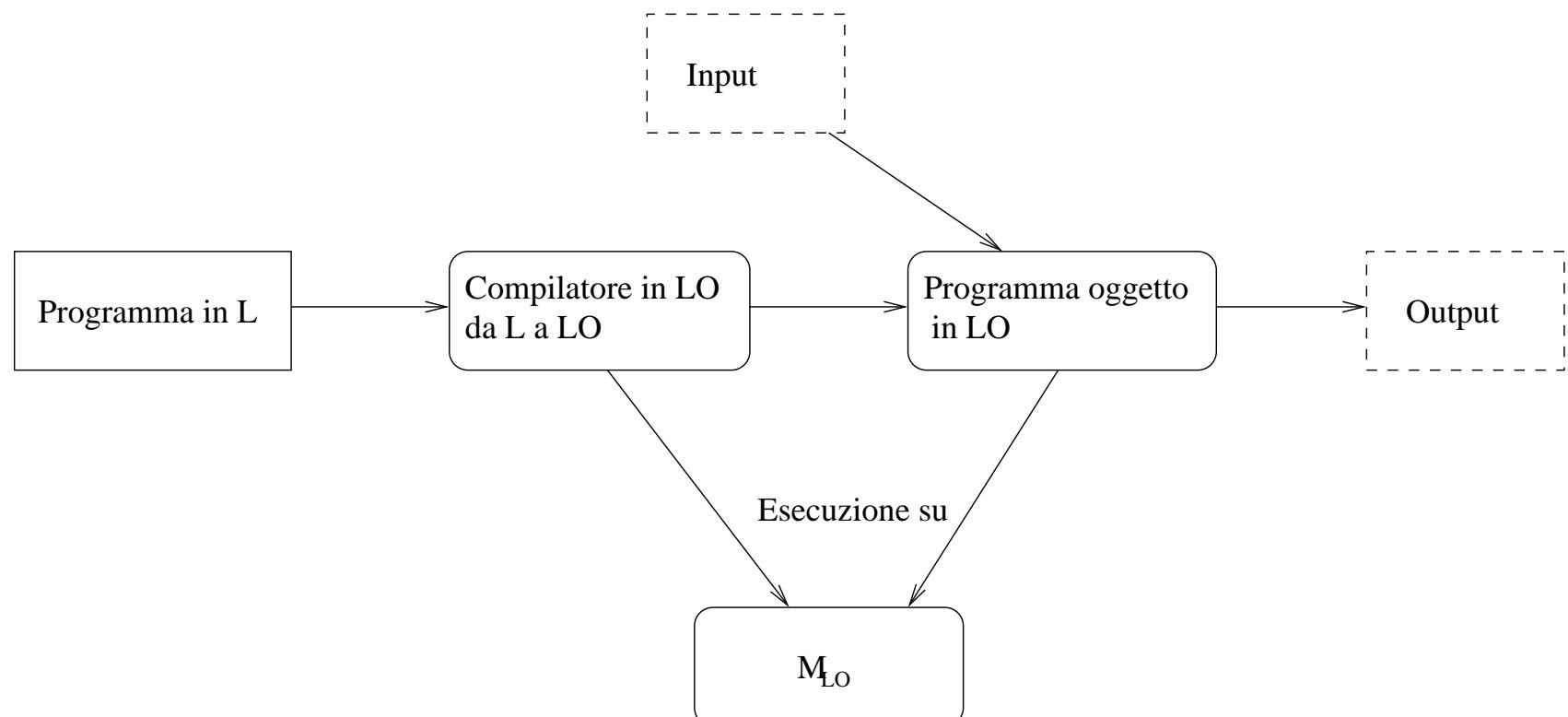
Compilatore

Componenti di un LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi computazionali





Compilazione pura (caso più generale)

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio → Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

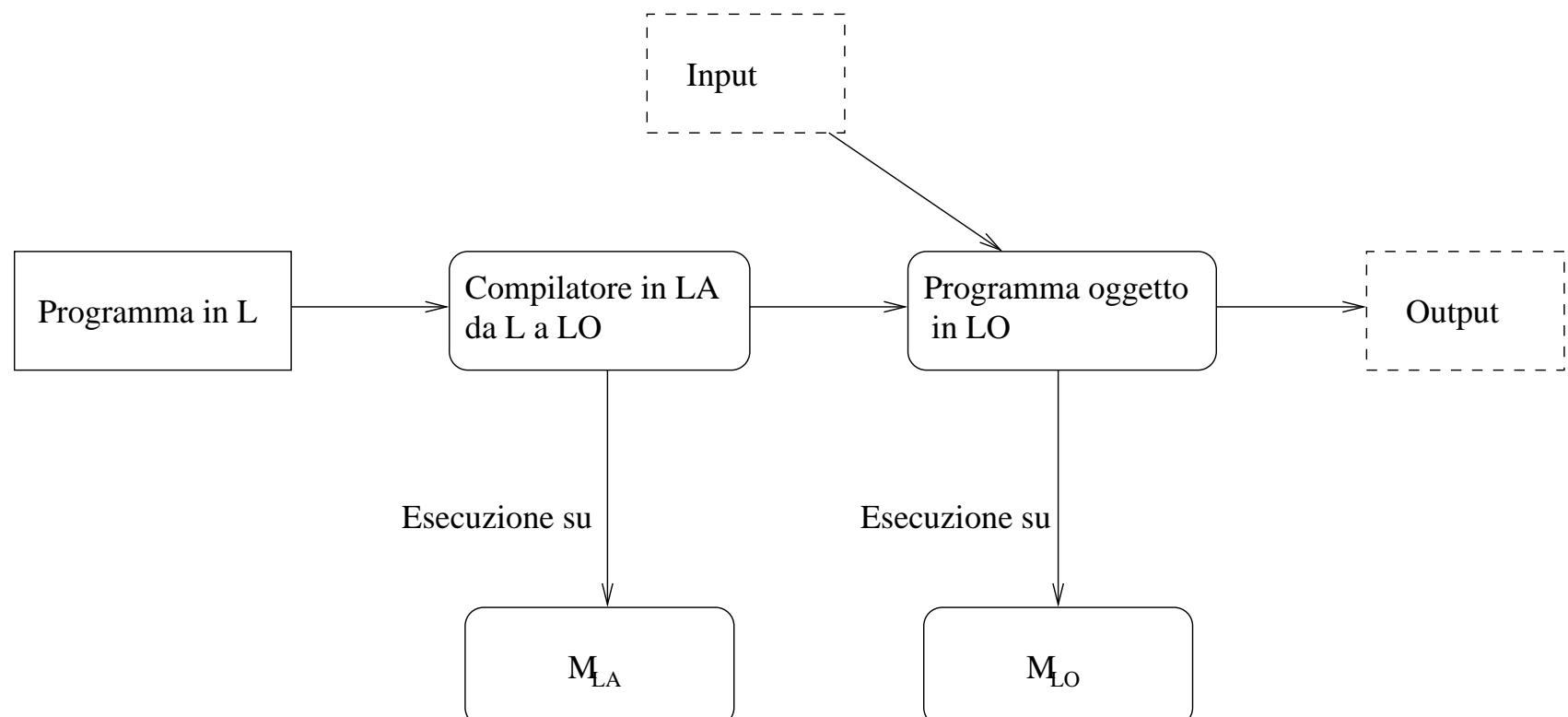
Compilatore

Componenti di un LP

Proprietà dei ...

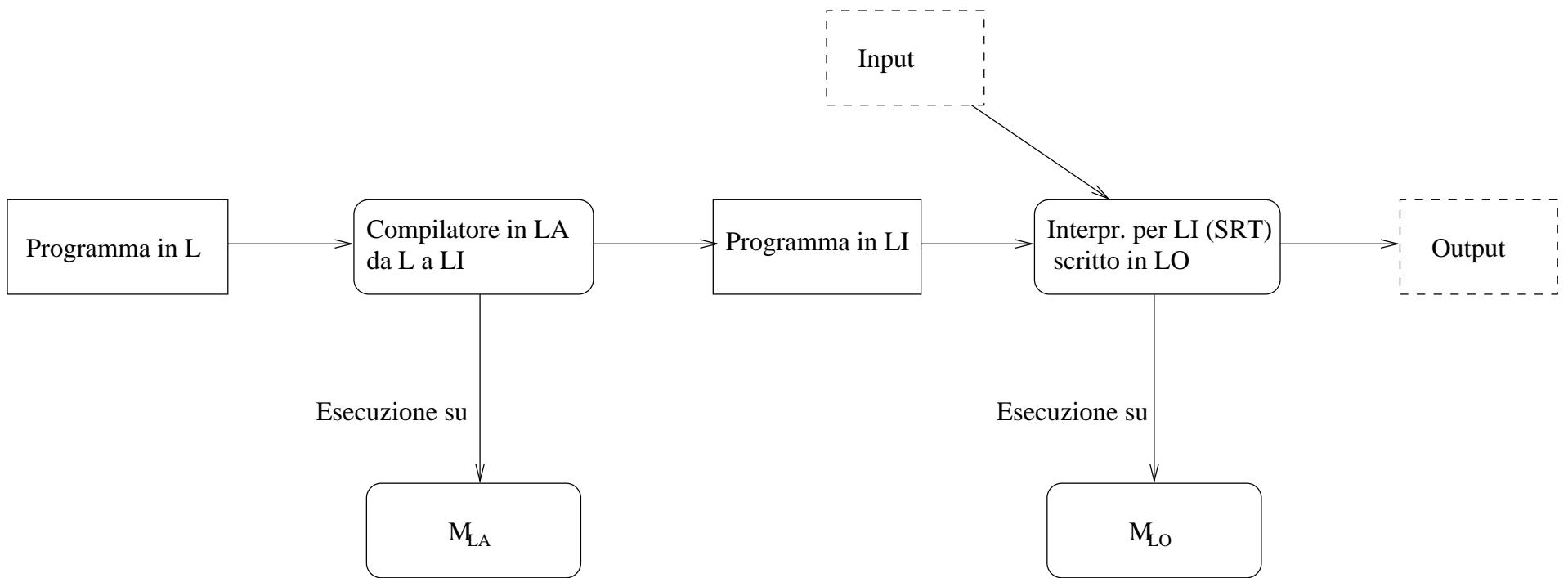
Criteri di scelta ...

Paradigmi computazionali





Compilazione per macchina intermedia





Compilazione ed esecuzione

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio → Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

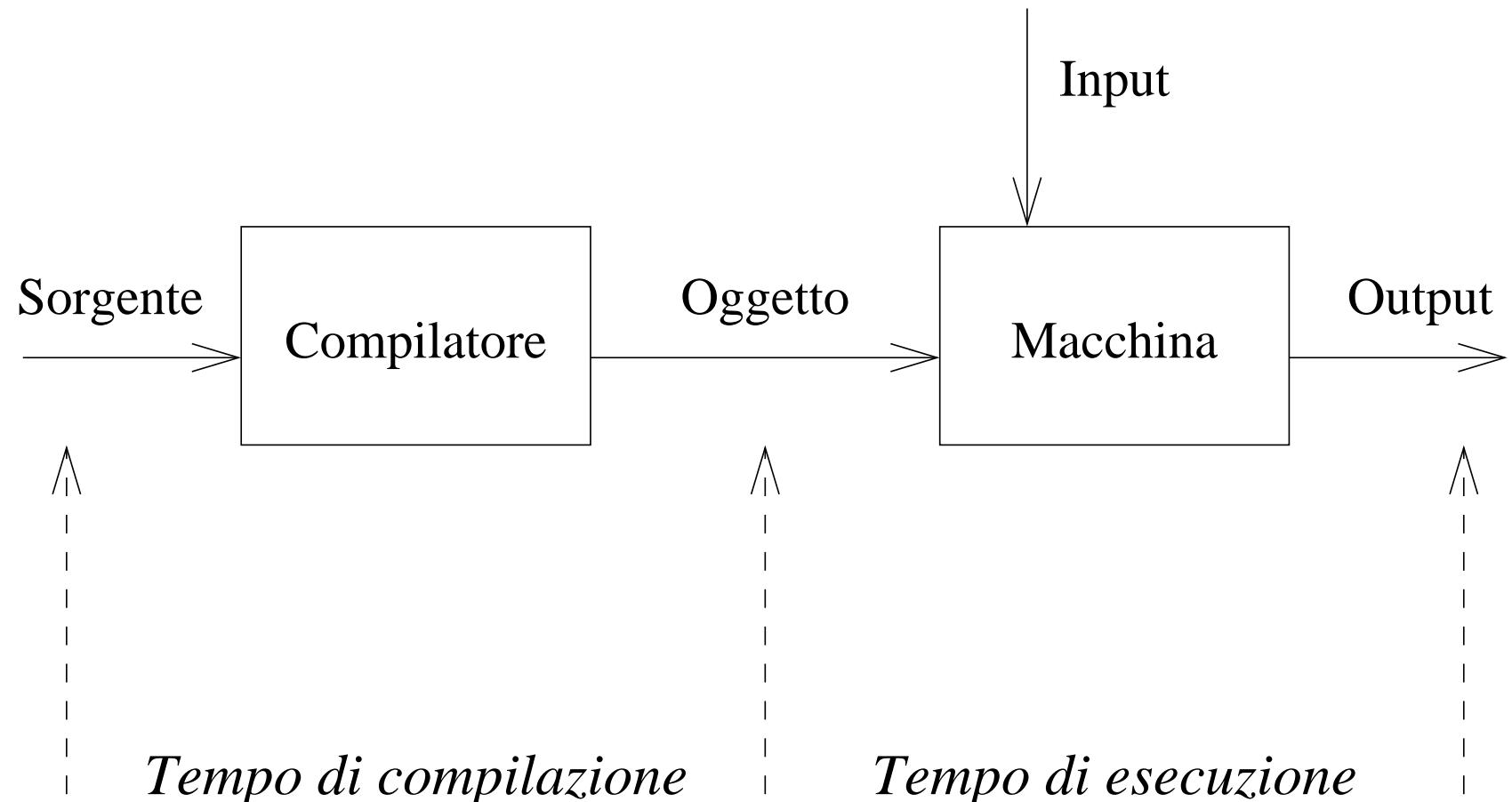
Compilatore

Componenti di un LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi computazionali





Compilatore

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

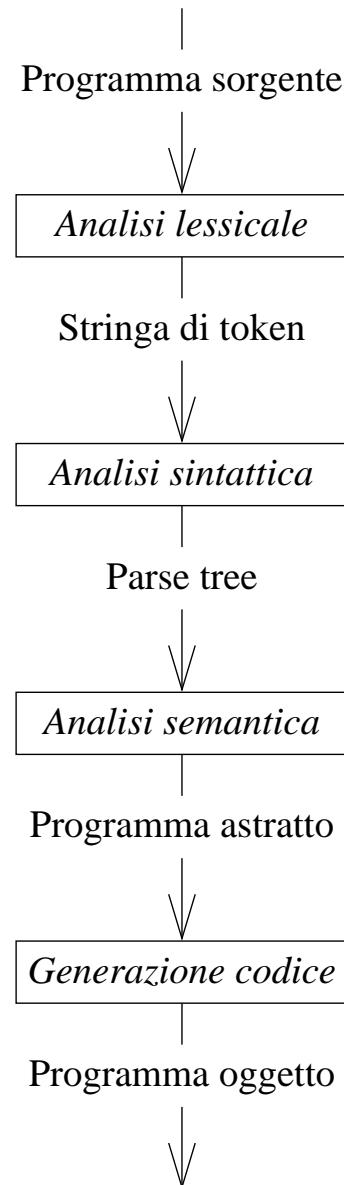
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali





Componenti di un LP

■ Grammatica

- ◆ *Come si scrive un programma ben formato (cioè, eseguibile)?*
- ◆ *Anche detta sintassi*

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Componenti di un LP

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali

■ Grammatica

- ◆ *Come si scrive un programma ben formato (cioè, eseguibile)?*

- ◆ Anche detta *sintassi*

■ Semantica

- ◆ *Qual è il significato di ciascuna istruzione/costrutto?*



Componenti di un LP

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

■ Grammatica

- ◆ *Come si scrive un programma ben formato (cioè, eseguibile)?*

- ◆ Anche detta *sintassi*

■ Semantica

- ◆ *Qual è il significato di ciascuna istruzione/costrutto?*

■ Pragmatica

- ◆ *Come si usa efficacemente il linguaggio? (Programmazione & Ingegneria del Software)*



Componenti di un LP

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

Grammatica

◆ *Come si scrive un programma ben formato (cioè, eseguibile)?*

◆ *Anche detta sintassi*

Semantica

◆ *Qual è il significato di ciascuna istruzione/costrutto?*

Pragmatica

◆ *Come si usa efficacemente il linguaggio? (Programmazione & Ingegneria del Software)*

Implementazione

◆ *Come si realizza una macchina astratta per questo linguaggio? (Costruzione di compilatori)*

In questo corso, ci concentreremo su grammatica e semantica, con accenni all'implementazione



Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

- Semplicità – (concisione) VS (leggibilità)
 - ◆ Sintattica: unica rappresentabilità di ogni concetto.
 - ◆ Semantica: minimo numero di concetti e strutture.



Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali

- Semplicità – (concisione) VS (leggibilità)
 - ◆ Sintattica: unica rappresentabilità di ogni concetto.
 - ◆ Semantica: minimo numero di concetti e strutture.
- Astrazione – (incapsulare e dividere)
 - ◆ Dati: nascondere i dettagli (incapsulamento, information hiding)
 - ◆ Funzioni e procedure: dividere il programma
 - ◆ Moduli, pacchetti, etc.: favorire la modularità



Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi

computazionali

- Semplicità – (concisione) VS (leggibilità)
 - ◆ Sintattica: unica rappresentabilità di ogni concetto.
 - ◆ Semantica: minimo numero di concetti e strutture.
- Astrazione – (incapsulare e dividere)
 - ◆ Dati: nascondere i dettagli (incapsulamento, information hiding)
 - ◆ Funzioni e procedure: dividere il programma
 - ◆ Moduli, pacchetti, etc.: favorire la modularità
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)



Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi computazionali

- Semplicità – (concisione) VS (leggibilità)
 - ◆ Sintattica: unica rappresentabilità di ogni concetto.
 - ◆ Semantica: minimo numero di concetti e strutture.
- Astrazione – (incapsulare e dividere)
 - ◆ Dati: nascondere i dettagli (incapsulamento, information hiding)
 - ◆ Funzioni e procedure: dividere il programma
 - ◆ Moduli, pacchetti, etc.: favorire la modularità
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)
- Ortogonalità – (poche eccezioni alle regole del linguaggio)



Proprietà dei linguaggi

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi computazionali

- Semplicità – (concisione) VS (leggibilità)
 - ◆ Sintattica: unica rappresentabilità di ogni concetto.
 - ◆ Semantica: minimo numero di concetti e strutture.
- Astrazione – (incapsulare e dividere)
 - ◆ Dati: nascondere i dettagli (incapsulamento, information hiding)
 - ◆ Funzioni e procedure: dividere il programma
 - ◆ Moduli, pacchetti, etc.: favorire la modularità
- Espressività – (facilità di rappresentazione di oggetti) VS (semplicità)
- Ortogonalità – (poche eccezioni alle regole del linguaggio)
- Portabilità



Criteri di scelta del linguaggio

■ Disponibilità di interpreti/compilatori

Introduzione al corso

Linguaggi (di programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Criteri di scelta del linguaggio

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un

LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Criteri di scelta del linguaggio

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità
- Interoperabilità con altri sistemi e linguaggi

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Criteri di scelta del linguaggio

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità
- Interoperabilità con altri sistemi e linguaggi
- Esistenza di librerie specifiche

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Criteri di scelta del linguaggio

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità
- Interoperabilità con altri sistemi e linguaggi
- Esistenza di librerie specifiche
- Maggiore conoscenza da parte del programmatore

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →
Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali



Criteri di scelta del linguaggio

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità
- Interoperabilità con altri sistemi e linguaggi
- Esistenza di librerie specifiche
- Maggiore conoscenza da parte del programmatore
- Comodità dell'ambiente di programmazione (IDE)
- Sintassi aderente al problema



Criteri di scelta del linguaggio

Introduzione al corso

Linguaggi (di
programmazione)

Macchine astratte

Definizione

Esempio

Processore M.A.

Esempi M.A.

Realizzazione M.A.

Traduttori

Linguaggio →

Macchina astratta

Interpretazione pura

Compilazione 1

Compilazione 2

Compilazione 3

Compilazione ed ...

Compilatore

Componenti di un
LP

Proprietà dei ...

Criteri di scelta ...

Paradigmi
computazionali

- Disponibilità di interpreti/compilatori
- Esistenza di standard di portabilità
- Interoperabilità con altri sistemi e linguaggi
- Esistenza di librerie specifiche
- Maggiore conoscenza da parte del programmatore
- Comodità dell'ambiente di programmazione (IDE)
- Sintassi aderente al problema



[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

**Paradigmi
computazionali**

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

Paradigmi computazionali



Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).



Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*.
Esecuzione come valutazione di funzioni.



Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*.
Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema.
Esecuzione analoga a processi di dimostrazione di teoremi.



Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*.
Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema.
Esecuzione analoga a processi di dimostrazione di teoremi.

Orientato ad oggetti: Programma costituito da oggetti che scambiano messaggi.



Paradigmi

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

[Paradigmi](#)

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

Imperativo: Un programma specifica sequenze di modifiche da apportare allo *stato della macchina* (memoria).

Funzionale: Il programma e le sue componenti sono *funzioni*.
Esecuzione come valutazione di funzioni.

Logico: Programma come descrizione logica di un problema.
Esecuzione analoga a processi di dimostrazione di teoremi.

Orientato ad oggetti: Programma costituito da oggetti che scambiano messaggi.

Parallello: Programmi che descrivono entità distribuite che sono eseguite contemporaneamente ed in modo asincrono.

Gli ultimi due sono ortogonali rispetto ai primi tre...



Esempio 1.0 Imperativo vs. Funzionale

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

**Esempio 1.0
Imperativo vs.
Funzionale**

[Esempio 1.1](#)

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Esempio 3](#)

[Conclusioni](#)

function definition

```
function factI (n)
    local accumulator = 1
    for i = 1,n do
        accum = accumulator*i
    end
    return accum
end
```

trace of execution

```
factI(4):
    accumulator = 1
    i = 1    accumulator = 1 * 1
    i = 2    accumulator = 1 * 2
    i = 3    accumulator = 2 * 3
    i = 4    accumulator = 6 * 4
    return 24
```

function definition

```
function factR (n)
    if n == 1 then
        return 1
    else
        return n*factR(n-1)
    end
end
```

trace of execution

```
factR(4) =
4 * factR(3) =
    3 * factR(2) =
        2 * factR(1) =
            1
                1
                    1
                        2
                            6
```

24

Notare eliminazione assegnamenti: n rimpiazza i , ricorsione invece di cicli



Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `member(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi
Esempio 1.0
Imperativo vs.
Funzionale](#)

[Esempio 1.1](#)

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Esempio 3](#)

[Conclusioni](#)



Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `member(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

```
member(2, [1, 2, 3]) = true;  
member(4, [1, 2, 3]) = false.
```

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi
Esempio 1.0
Imperativo vs.
Funzionale](#)

Esempio 1.1

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Esempio 3](#)

[Conclusioni](#)



Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `member(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

```
member(2, [1, 2, 3]) = true;
```

```
member(4, [1, 2, 3]) = false.
```

A questo scopo, si suppongano già esistenti le funzioni:

- `empty(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

[Esempio 1.0](#)

[Imperativo vs.](#)

[Funzionale](#)

Esempio 1.1

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Esempio 3](#)

[Conclusioni](#)



Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `member(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

```
member(2, [1, 2, 3]) = true;  
member(4, [1, 2, 3]) = false.
```

A questo scopo, si suppongano già esistenti le funzioni:

- `empty(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.
- `first(L)`, che restituisce il primo elemento della lista `L`.
Es.: `first([1, 2, 3]) = 1.`

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi](#)

[Esempio 1.0](#)

[Imperativo vs.](#)

[Funzionale](#)

Esempio 1.1

[Esempio 1.2](#)

[Esempio 2](#)

[Esempio 3](#)

[Esempio 3](#)

[Conclusioni](#)



Esempio 1.1

Vogliamo scrivere in un linguaggio imperativo, funzionale e logico la funzione `member(X, L)` che decida se l'elemento `X` appartiene alla lista `L`.

Es.:

`member(2, [1, 2, 3]) = true;`

`member(4, [1, 2, 3]) = false.`

A questo scopo, si suppongano già esistenti le funzioni:

- `empty(L)`, che restituisce `true` se `L` è vuota, altrimenti `false`.
- `first(L)`, che restituisce il primo elemento della lista `L`.
Es.: `first([1, 2, 3]) = 1.`
- `rest(L)`, che restituisce una sottolista ottenuta rimuovendo il primo elemento di `L`.
Es.: `rest([1, 2, 3]) = [2, 3].`



Esempio 1.2

La funzione member nel paradigma imperativo (pseudo-codice):

```
procedure member(X,L)
    local L1 = L
    while not empty(L1) and X != first(L1)
        do L1 = rest(L1)
    return not empty(L1)
```

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi
Esempio 1.0
Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 1.2

La funzione member nel paradigma imperativo (pseudo-codice):

```
procedure member(X,L)
    local L1 = L
    while not empty(L1) and X != first(L1)
        do L1 = rest(L1)
    return not empty(L1)
```

In C (un concreto linguaggio imperativo):

```
bool member(int X, Lista L) {
    Lista L1 = L;
    while( ! empty(L1) && X != first(L1))
        L1 = rest(L1);
    return ! empty(L1);
}
```

NB: nessuna differenza strutturale, solo dettagli sintattici



Esempio 2

La funzione member nel paradigma funzionale (come Lisp):

```
function member(X,L)
  if empty(L)  then false
  else if X == first(L) then true
        else member(X, rest(L))
```

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 2

La funzione member nel paradigma funzionale (come Lisp):

```
function member(X,L)
  if empty(L)  then false
  else if X == first(L) then true
  else member(X, rest(L))
```

Nella versione funzionale pura non ci sono variabili, nè assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione. La sintassi Lisp e Scheme sarebbe un po' particolare:

```
(defun member (x l)
  (cond ((null l) nil)
        ((equal x (first l)) T)
        (T (member x (rest l)))))
```

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 2

La funzione member nel paradigma funzionale (come Lisp):

```
function member(X,L)
  if empty(L)  then false
  else if X == first(L) then true
  else member(X, rest(L))
```

Nella versione funzionale pura non ci sono variabili, nè assegnazioni. Quindi non si possono usare cicli e bisogna rimpiazzarli con la ricorsione. La sintassi Lisp e Scheme sarebbe un po' particolare:

```
(defun member (x l)
  (cond ((null l) nil)
        ((equal x (first l)) t)
        (t (member x (rest l)))))
```

Anche il C si potrebbe usare in stile funzionale se evitassimo di usare i costrutti imperativi:

```
bool member(int X, Lista L) {
    return (empty(L))? false : (
        (X == first(L))? true :
        member(X, rest(L)) ) }
```



Esempio 3

La funzione `member` nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

- `[X|L]` è un *pattern* che denota una lista in cui `X` è la testa e `L` è il resto
- `member` è un *predicato*
- I parametri formali dei predicati possono essere pattern
- I programmi consistono di definizioni di predicati

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

[Paradigmi Esempio 1.0](#)

[Imperativo vs. Funzionale](#)

[Esempio 1.1](#)

[Esempio 1.2](#)

[Esempio 2](#)

Esempio 3

[Esempio 3](#)

[Conclusioni](#)



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- `member(2, [1,2,3])` restituisce *yes* (`true`)
- `member(0, [1,2,3])` restituisce *no* (`false`)

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi
Esempio 1.0
Imperativo vs.

Funzionale
Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (*true*)
- member(0, [1,2,3]) restituisce *no* (*false*)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi
Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

[Introduzione al corso](#)

[Linguaggi \(di
programmazione\)](#)

[Macchine astratte](#)

[Paradigmi
computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (true)
- member(0, [1,2,3]) restituisce *no* (false)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1; X=2



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (*true*)
- member(0, [1,2,3]) restituisce *no* (*false*)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1; X=2; X=3

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi
Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (*true*)
- member(0, [1,2,3]) restituisce *no* (*false*)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1; X=2; X=3; no (si comporta come un generatore)

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (*true*)
- member(0, [1,2,3]) restituisce *no* (*false*)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1; X=2; X=3; no (si comporta come un generatore)
- Risposte con variabili: member(1,L) restituisce
 - ◆ L=[1 | L₀]

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (*true*)
- member(0, [1,2,3]) restituisce *no* (*false*)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1; X=2; X=3; no (si comporta come un generatore)
- Risposte con variabili: member(1,L) restituisce
 - ◆ L=[1|L₀]; L=[Y₀,1|L₁]

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- member(2, [1,2,3]) restituisce *yes* (*true*)
- member(0, [1,2,3]) restituisce *no* (*false*)
- Query con variabili: member(X,[1,2,3]) restituisce
 - ◆ X=1; X=2; X=3; no (si comporta come un generatore)
- Risposte con variabili: member(1,L) restituisce
 - ◆ L=[1|L₀]; L=[Y₀,1|L₁]; L=[Y₀,Y₁,1|L₂] ...

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Esempio 3

La funzione member nel paradigma logico (prolog):

```
member(X, [X|L]).  
member(X, [Y|L]) :- member(X, L).
```

Esecuzione:

- `member(2, [1,2,3])` restituisce *yes* (`true`)
- `member(0, [1,2,3])` restituisce *no* (`false`)
- Query con variabili: `member(X,[1,2,3])` restituisce
 - ◆ `X=1; X=2; X=3; no` (si comporta come un generatore)
- Risposte con variabili: `member(1,L)` restituisce
 - ◆ `L=[1|L0]; L=[Y0,1|L1]; L=[Y0,Y1,1|L2] ...`
- *Invertibilità*: nessuna distinzione tra input e output. Un solo predicato (programma), molte funzioni.

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi

Esempio 1.0

Imperativo vs.

Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni



Conclusioni

[Introduzione al corso](#)

[Linguaggi \(di programmazione\)](#)

[Macchine astratte](#)

[Paradigmi computazionali](#)

Paradigmi
Esempio 1.0
Imperativo vs.
Funzionale

Esempio 1.1

Esempio 1.2

Esempio 2

Esempio 3

Esempio 3

Conclusioni

- Il paradigma di appartenenza può influenzare *radicalmente* il modo in cui si risolve il problema
 - ◆ In linguaggi dello stesso paradigma, lo stesso problema ha soluzioni strutturalmente identiche
 - ◆ Imparato a risolvere un problema in un linguaggio, lo si sa risolvere in tutti i linguaggi dello stesso paradigma
- Il paradigma non è l'unico aspetto determinante. Altri esempi di aspetti importanti:
 - ◆ Il sistema di tipi supportato
 - ◆ Eventuale supporto alle eccezioni
 - ◆ Modello di concorrenza e sincronizzazione
 - ◆ Presenza di garbage collection
 - ◆ ...

Java: Costruzione di oggetti

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli "Federico II"

Corso di Linguaggi di Programmazione I

Elementi sintattici:

- I costruttori (metodi col nome della classe e senza tipo di ritorno)
- Invocazioni da un costruttore a un altro (this e super)
- I blocchi di inizializzazione (blocchi di codice anonimi nello scope di classe)
- Gli inizializzatori

```
public class A {  
    private int n = <exp>;  
  
    public A(int a) {  
        n = a;  
    }  
  
    {  
        n++;  
    }  
}
```

Inizializzatore

Costruttore

Blocco di
inizializzazione

Invocazioni **esplicite** ad un altro costruttore:

- Un costruttore può chiamarne un altro della stessa classe usando la parola chiave **this**, oppure un costruttore della sua superclasse diretta usando la parola chiave **super**
- *this* e *super*, usati in questa accezione, devono comparire alla **prima riga** del costruttore, pena un errore di compilazione
- Attenzione: ricordate che *this* e *super* hanno anche altri significati:
 - *this* rappresenta il riferimento all'oggetto corrente
 - *super* si usa per riferirsi a un elemento (metodo o attributo) *mascherato* appartenente a una superclasse
 - Ad esempio, per invocare la versione originale di un metodo di cui stiamo facendo l'overriding

Esempio 1:

L'utente sceglie se fornire un valore o usare quello di default

```
class A {  
    private int size;  
    public A() {  
        this(1000); // invoca l'altro costruttore  
    }  
    public A(int n) {  
        size = n;  
    }  
}
```

Esempio 2:

```
class A {  
    public A() { this("Costruttore senza argomenti"); }  
    public A(String msg) { System.out.println(msg); }  
}  
  
class B extends A {  
    public B() { }  
    public B(int n) {  
        super("Valore: " + n);  
    }  
}
```

Chiamate **implicite** ad un altro costruttore:

- Se un costruttore **non inizia** con una chiamata ad un altro costruttore (*this* o *super*), il compilatore inserisce **automaticamente** una chiamata al costruttore senza argomenti della superclasse
 - Ovvero, inserisce l'istruzione “*super()*”
- In tal caso, se la superclasse non ha un costruttore senza argomenti, si verifica un errore di compilazione

Il meccanismo tramite il quale i costruttori possono invocarsi a vicenda prende il nome di **concatenazione dei costruttori** (*constructor chaining*)

Sequenza di inizializzazione di un oggetto:

- 1a) Se il costruttore inizia con super(...), passare al costruttore della superclasse (risoluzione overloading)
- 1b) Se il costruttore inizia con this(...), passare al costruttore indicato (risoluzione overloading)
- 1c) Se il costruttore non inizia né con super né con this, passare al costruttore senza argomenti della superclasse (a meno di non essere in Object)
- 2) Eseguire i *blocchi di inizializzazione* e gli *inizializzatori* degli attributi, nell'ordine in cui compaiono nel codice
- 3) Eseguire il resto del costruttore

Nota 1: un attributo privo di inizializzatore assume il valore di default del suo tipo

Nota 2: un blocco di inizializzazione o un inizializzatore non può fare riferimento a un attributo la cui dichiarazione non

- Il compilatore controlla che la concatenazione **non sia ciclica**
- Infatti, se dei costruttori si chiamano a vicenda, siccome tali chiamate si trovano alla prima riga del rispettivo costruttore, e pertanto avvengono incondizionatamente, ci si trova in presenza di una *mutua ricorsione non ben fondata*, ovvero infinita
- Ad esempio, tentando di compilare la seguente classe:

```
class A {  
    public A() { this(3); }  
    public A(int i) { this(); }  
}
```

- si ottiene il seguente errore di compilazione:

A.java:3: recursive constructor invocation

```
public A(int i) { this(); }  
          ^
```

Per analizzare la concatenazione dei costruttori di una data gerarchia di classi, ed in particolare controllare che essa non sia ciclica, è possibile realizzare il seguente diagramma:

- 1) Creare un **grafo** con un **nodo** per ogni costruttore, compresi quelli impliciti
- 2) Aggiungere un **arco** da un nodo x ad un nodo y se il costruttore x chiama, esplicitamente o meno, il costruttore y
- 3) Il grafo ottenuto non deve presentare cicli

Determinare l'output del seguente programma

```
public class A {  
    public A() {  
        System.out.println("A()");  
    }  
}  
  
public class B extends A {  
    public B() {  
        this(0);  
        System.out.println("B()");  
    }  
    public B(int n) {  
        System.out.println("B(int)");  
    }  
    public static void main(String[] args) {  
        new B();  
    }  
}
```

Determinare l'output del seguente programma

```
public class A {  
  
    public A() {  
        System.out.println("A()");  
    }  
  
    {  
        System.out.println("Blocco 1");  
    }  
  
}
```

```
public class B extends A {  
  
    public B() {  
        this(0);  
        System.out.println("B()");  
    }  
  
    {  
        System.out.println("Blocco 2");  
    }  
  
    public B(int n) {  
        System.out.println("B(int)");  
    }  
  
    public static void main(String[] args)  
    {  
        new B();  
    }  
  
}
```

- Esercizio d'esame 27/3/08, #4

Java: Il sistema dei tipi

I tipi wrapper

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione I

- Il linguaggio Java è **staticamente tipizzato**
- "tipato" vuol dire che ad ogni espressione viene assegnato un tipo, che rappresenta l'insieme di valori che l'espressione potrebbe assumere
- "staticamente" vuole dire che il tipo di ogni espressione deve essere noto al momento della compilazione
- In tal modo, il **compilatore** è in grado di controllare che tutte le operazioni (compresa, ad esempio, l'assegnazione) siano applicate ad operandi **compatibili**
- Questa fase della compilazione è chiamata appunto "*type checking*"

- I **tipi base** sono otto:
 - boolean
 - char
 - byte, short, int, long
 - float e double
- Inoltre, void è un tipo speciale usato solo come tipo di ritorno dai metodi
- Tra tipi base esistono le seguenti **conversioni implicite** (o **promozioni**):
 - Da byte a short, da short a int, da int a long, da long a float e da float a double
 - Da char a int
- Le conversioni implicite sono transitive
- Se esiste una conversione implicita dal tipo x al tipo y, è possibile assegnare un valore di tipo x ad una variabile di tipo y (ci torneremo con la *relazione di assegnabilità*)
- Si noti come non ci sono conversioni da e per il tipo boolean

Alcune conversioni implicite possono comportare una *perdita di informazione*

- Ad esempio:

```
int i = 1000000001; // un miliardo e uno
float f = i;
```

Dopo queste istruzioni, f contiene un miliardo, perché un float *non ha abbastanza bit di mantissa* per rappresentare le 10 cifre significative di i

Le conversioni a rischio sono quella da int a float e, a maggior ragione, da long a float

Inserire una dichiarazione e inizializzazione per la variabile x, in modo che il seguente ciclo sia infinito:

```
while (x == x+1) {  
    ...  
}
```

- A differenza di altri linguaggi orientati agli oggetti (ad es., il C++), non esistono variabili che contengono oggetti, solo *riferimenti* ad oggetti, ovvero variabili che contengono l'*indirizzo* di un oggetto
- I riferimenti Java sono quindi simili ai puntatori del linguaggio C
- Tuttavia, i riferimenti Java sono molto più restrittivi
 - Niente aritmetica dei puntatori (`p++`) e conversioni tra puntatori e numeri interi
 - Niente doppi puntatori
 - Niente puntatori a funzioni
- Quindi, rispetto ai puntatori, i riferimenti Java sono meno potenti, più facili da utilizzare e meno soggetti ad errori al run-time
- Java prevede solo il passaggio per *valore*: sia i tipi base che i riferimenti sono passati per valore
- Non è possibile passare oggetti per valore, l'unico modo di manipolare (ed in particolare, passare) oggetti è tramite i loro riferimenti (ovvero, indirizzi)

- In virtù del polimorfismo, bisogna distinguere due tipi di ogni variabile ed espressione di categoria “riferimento”:

```
Animal a = new Dog("Lilli");
```

- Animal è il tipo **dichiarato** (o statico) del riferimento “a”
- Il tipo dichiarato è noto a tempo di compilazione e rimane invariato in tutto l’ambito di validità di quel riferimento
- Dog è il tipo **effettivo** (o dinamico) del riferimento “a”, in questo momento
- Il tipo effettivo non è noto a tempo di compilazione, nel senso che il compilatore non tenta di determinarlo, perché sarebbe in generale impossibile (si pensi a un parametro di un metodo)
- Il tipo effettivo può cambiare ogni volta che si assegna un nuovo valore a quel riferimento

- Java prevede i seguenti tipi:
 - Gli 8 tipi base, o *primitivi*
 - I tipi riferimento
 - Tra cui: i tipi array
 - Il tipo speciale “nullo”
- I tipi base sono già stati introdotti
- I tipi riferimento corrispondono a classi, interfacce, enumerazioni, o array
- I tipi array (caso particolare di riferimento) sono tipi composti, ovvero si definiscono a partire da un altro tipo, detto tipo “componente”, e si individuano sintatticamente per l'uso delle parentesi quadre
- Il tipo nullo prevede come unico valore possibile la costante “null”

- Proviamo a simulare la fase di *type checking* del compilatore Java sul seguente frammento di codice:

```
int n;  
double d;  
  
d = (new Object()).hashCode() + n/2.0;
```

- Proviamo a simulare la fase di type checking del compilatore Java sul seguente frammento di codice:

```
int n;  
double d;  
  
d = (new Object()).hashCode() + n/2.0;
```

- Cominciamo dalla parte destra dell'assegnazione e procediamo dalle sottoespressioni più piccole via via fino all'intera parte destra
- La prima espressione base che incontriamo è “new Object()”
 - questa espressione è per definizione di tipo “riferimento alla classe Object”, o per brevità, di tipo “Object”
- Passiamo poi all'espressione “(new Object()).hashCode()”
 - in questo contesto il punto denota la chiamata ad un metodo
 - quindi, il tipo dell'espressione è il **tipo di ritorno** del metodo, in questo caso “int”
- Esercizio: quali altri significati del punto ci sono in Java?

```
int n;  
double d;  
  
d = (new Object()).hashCode() + n/2.0;
```

- L'espressione "n" ha chiaramente tipo "int", mentre "2.0" è una costante di tipo "double"
- Quindi, i due operandi della divisione "n/2.0" hanno tipo diverso
 - il primo operando viene quindi promosso da int a double tramite conversione implicita (si veda la lezione 1)
 - complessivamente, la divisione ha a sua volta tipo double
- Infine, per lo stesso motivo la somma avrà tipo double
- A questo punto, il compilatore verifica che il tipo calcolato per il lato destro dell'assegnazione sia compatibile (si veda dopo) con il tipo del lato sinistro (d)
- In questo caso, i due tipi coincidono esattamente
- Per completezza, anche l'intera assegnazione ha tipo "double"; questo consente di concatenare le assegnazioni, come in `a=b=c`
- Le regole di type checking sono specificate nel capitolo 15 della definizione del linguaggio Java (Java Language Specification)

- Per permettere il polimorfismo (in particolare, la capacità di un riferimento di puntare ad oggetti di tipo diverso), esiste una relazione binaria tra tipi, chiamata **relazione di sottotipo**
- La relazione di sottotipo è definita dalle seguenti regole, in cui T ed U rappresentano tipi arbitrari, *esclusi i tipi base*:
 - 1) T è sottotipo di se stesso
 - 2) T è sottotipo di Object
 - 3) Se T estende U oppure implementa U, T è sottotipo di U
 - 4) Il tipo nullo è sottotipo di T
 - 5) Se T è sottotipo di U allora T[] è sottotipo di U[]

- Per permettere il polimorfismo (in particolare, la capacità di un riferimento di puntare ad oggetti di tipo diverso), esiste una relazione binaria tra tipi, chiamata **relazione di sottotipo**
- La relazione di sottotipo è definita dalle seguenti regole, in cui T ed U rappresentano tipi arbitrari, *esclusi i tipi base*:
 - 1) T è sottotipo di se stesso
 - 2) T è sottotipo di Object
 - 3) Se T estende U oppure implementa U, T è sottotipo di U
 - 4) Il tipo nullo è sottotipo di T
 - 5) Se T è sottotipo di U allora T[] è sottotipo di U[]
- La relazione di sottotipo non coinvolge i tipi base
- E' facile verificare che la relazione di sottotipo è riflessiva, antisimmetrica e transitiva
 - pertanto, essa è una relazione d'ordine sull'insieme dei tipi (non base)
- Queste regole non tengono conto dei tipi parametrici introdotti da Java 1.5, di cui si parlerà nelle successive lezioni

La relazione di sottotipo permette di definire precisamente il comportamento dell'operatore *instanceof*

Data un'espressione *exp* ed il nome di una classe o interfaccia *T*, l'espressione

exp instanceof T

restituisce *vero* se e solo se il tipo **effettivo** di *exp* **non è nullo ed è sottotipo di T**

Nota: il primo argomento di *instanceof* deve essere un'espressione di categoria "riferimento", pena un errore di compilazione

- La relazione di *compatibilità* (o *assegnabilità*) tra tipi stabilisce quando è possibile assegnare un valore di un certo tipo T ad una variabile di tipo U
- Si dice che T è **assegnabile** ad U se
 - T è sottotipo di U, oppure
 - T ed U sono tipi base e c'è una conversione implicita da T ad U

La relazione di assegnabilità si applica nei seguenti contesti:

- Assegnazione: $a = \text{exp}$
- Chiamata a metodo: $x.f(\text{exp})$
- Ritorno da metodo: return exp

- In base alla definizione di sottotipo, un array di qualunque tipo è sottotipo di “array di Object”
- Questo consente, ad esempio, di passare qualunque array ad un metodo che abbia come parametro formale un array di Object
- Consideriamo il seguente esempio:

```
String[] arr1 = new String[10];
Object[] arr2 = arr1;
arr2[0] = new Object();
String s = arr1[0];
```

```
String[] arr1 = new String[10];
Object[] arr2 = arr1;
arr2[0] = new Object();
String s = arr1[0];
```

- L'esempio risulta corretto per il compilatore
- Tuttavia, nell'ultima istruzione assegnamo alla variabile "s", dichiarata String, un oggetto di tipo effettivo "Object", che non è compatibile
- In effetti, al run-time viene sollevata un'**eccezione** (ArrayStoreException) al momento della **terza** istruzione
- Questo perché al run-time gli array "ricordano" il tipo con il quale sono stati creati
- La JVM utilizza questa informazione per controllare che gli oggetti inseriti nell'array siano sempre di tipo compatibile con quello dichiarato in origine

I cast

- Java permette alcune conversioni esplicite di tipo tramite *cast*
 - Anche dette *coercizioni di tipo*
- La sintassi è:
$$(T) \ exp$$

- Si può utilizzare un cast per effettuare esplicitamente una **promozione**
 - in questo caso il cast è superfluo
- Si può utilizzare un cast per effettuare una **promozione al contrario**
 - ad esempio, da double a int
 - in questi casi, è facile incorrere in *perdite di informazioni*
 - ad esempio, nel passaggio da numeri in virgola mobile a numeri interi, si può perdere **sia in precisione che in magnitudine** (ordine di grandezza)
 - i dettagli sono definiti nella sezione 5.1.3 del JLS: Narrowing Primitive Conversion
 - queste conversioni sono decisamente sconsigliate, al loro posto è opportuno utilizzare i metodi appositi della classe Math (come Math.round)

- Sono **consentiti** dal compilatore i seguenti cast tra un tipo riferimento (o array) A ad un tipo riferimento (o array) B:
 - 1) se B è **supertipo** di A
 - si chiama “upcast”
 - è superfluo, perché i valori di tipo A sono di per sé assegnabili al tipo B
 - 2) se B è **sottotipo** di A
 - si chiama “downcast”
 - al run-time, la JVM controlla che l'oggetto da convertire appartenga effettivamente ad una sottoclasse di B
 - in caso contrario, viene sollevata l'eccezione ClassCastException
 - si deve cercare di evitare i downcast, perché aggirano il type checking svolto dal compilatore
 - a tale scopo, i tipi parametrici introdotti da Java 1.5 possono aiutare
 - se proprio si deve usare un downcast, esso andrebbe preceduto da un controllo instanceof, che assicuri la correttezza della conversione
- Negli altri casi, il cast porta a un errore di compilazione

Sapendo che sia la classe C sia la classe B estendono A, effettuare il type checking del seguente codice, evidenziando:

- errori di tipo
- cast non validi
- cast validi ma potenzialmente pericolosi al run-time

```
boolean f(A a, B b) {  
    C c = (C) a;  
    A a1 = (A) b;  
    Object o = a;  
    A[] arr = new A[10];  
    arr[5] = (Object) a;  
    arr[6] = b;  
    return a == c;  
}
```

I tipi Wrapper

- Per ogni tipo base, Java offre una corrispondente classe, che ingloba (*wraps*) un valore di quel tipo in un oggetto
- Queste classi, dette appunto wrapper, servono a trattare i valori base come se fossero oggetti
 - ad esempio, per inserirli nelle strutture dati offerte dalla Java Collection Framework (vedi lezioni seguenti)
- Le classi wrapper sono:
 - Byte, Short, Integer, Long, Float, Double
 - Boolean
 - Character
 - Void
- Come si vede, sono tutte omonime del rispettivo tipo base, tranne Integer, Character, e Void (perché formalmente void non è un tipo base)

Caratteristiche base delle classi wrapper

- Tutte le classi wrapper sono **immutabili** e **final** (come le stringhe)
I
- **Immutabile:** non è possibile modificare il contenuto di un oggetto
- **Final:** non è possibile creare sottoclassi

I

- Ogni classe wrapper ha un metodo statico **valueOf** che prende come argomento un valore del tipo base corrispondente alla classe e restituisce un oggetto wrapper che lo ingloba
- Esempio:

```
Integer n = Integer.valueOf(3);  
Double x = Double.valueOf(3.1415);
```

- A differenza di un costruttore, l'oggetto restituito da valueOf *non è necessariamente nuovo*
 - ovvero, il metodo valueOf cerca di riciclare gli oggetti già creati (*caching*)
 - questo non è un problema, perché gli oggetti wrapper sono immutabili

- Le sei classi wrapper relative ai tipi numerici estendono la **classe astratta Number**
- La classe Number prevede sei metodi, che estraggono il valore contenuto, convertendolo nel tipo base desiderato

```
public byte    byteValue()
public short   shortValue()
public int     intValue()
public long    longValue()
public float   floatValue()
public double  doubleValue()
```

- Se un oggetto wrapper viene convertito in un valore base verso il quale non c'è una conversione implicita (ad es., da double a int), l'effetto sarà lo stesso di quello di un cast

- Fino alla versione 1.4 di Java, era necessario convertire esplicitamente i valori dei tipi base in oggetti e viceversa
- A partire dalla versione 1.5, questo procedimento è stato automatizzato, introducendo **l'autoboxing e l'auto-unboxing**
- Grazie a queste funzionalità, il compilatore si occupa di inserire le istruzioni di conversione laddove queste siano necessarie

- Ad esempio, l'istruzione

```
Integer n = 7;
```

viene convertita in:

```
Integer n = Integer.valueOf(7);
```

- Allo stesso modo, le istruzioni:

```
Integer n = 7;
```

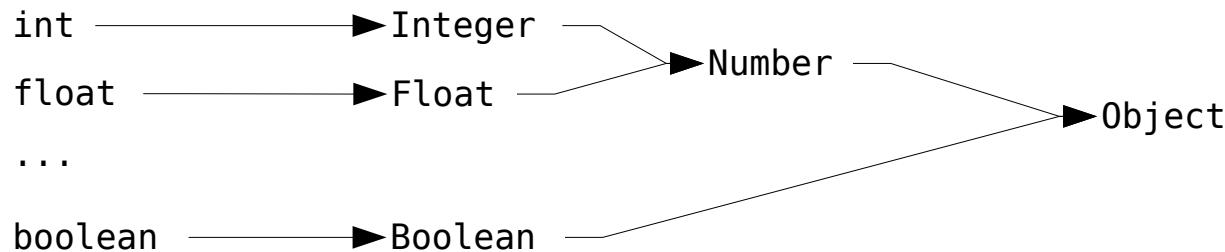
```
Integer i = n + 7;
```

vengono convertite in:

```
Integer n = Integer.valueOf(7);
```

```
Integer i = Integer.valueOf(n.intValue() + 7);
```

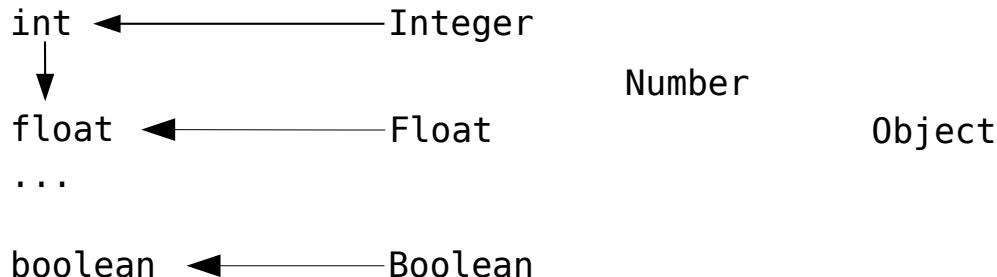
- L'autoboxing può convertire un'espressione di tipo primitivo in un oggetto del tipo wrapper corrispondente, o di un suo supertipo
- Quindi, l'autoboxing può effettuare le seguenti trasformazioni:



- L'autoboxing *non* prende in considerazione le conversioni implicite
- Ad esempio, ciascuna delle seguenti istruzioni provoca un **errore di compilazione**, perché oltre all'autoboxing richiederebbe anche una conversione implicita di tipo (promozione):

```
Double x = 1;  
Double y = 1.0f;  
Integer n = (byte) 1;
```

- L'auto-unboxing può essere seguito da una promozione



- Ad esempio, le seguenti istruzioni sono corrette:

```
Integer i = 7;      // autoboxing
double d = i;       // auto-unboxing seguito da promozione
```

- L'auto-unboxing di un'espressione che a runtime risulta **null** comporta il lancio di un'eccezione (verificata o non verificata?)

- L'operatore “==” può dare risultati sorprendenti se applicato ai tipi wrapper
- Infatti, è facile dimenticare che si tratta di un **confronto tra riferimenti**, come per tutti gli oggetti
- Ad esempio:

```
Integer a = new Integer(7), b = new Integer(7);  
System.out.println(a==b); // false, sono oggetti diversi
```

```
Integer a = 7, b = 7;  
System.out.println(a==b); // true, perché viene chiamato il metodo statico valueOf,  
                         che riutilizza gli oggetti corrispondenti a valori piccoli
```

```
Integer a = 700, b = 700;  
System.out.println(a==b); // false, perché il metodo valueOf riutilizza soltanto  
                         gli interi compresi tra -127 e 127
```

- Morale: confrontare i tipi wrapper con **equals** e non con “==”

Java: Risoluzione dell'overloading e dell'overriding

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli "Federico II"

Corso di Linguaggi di Programmazione I

- Per “binding dinamico” (letteralmente, “bind” significa “legare”) si intende il meccanismo per cui non è il compilatore, ma la JVM ad avere l’ultima parola su **quale metodo invocare** in corrispondenza di ciascuna chiamata a metodo
- In altri termini, si tratta di stabilire il *legame di locazione* di un nome di metodo nel contesto di una invocazione

- Esempio:

$x.f(exp)$

- Il binding di x è statico
 - L’identità di x viene stabilita a tempo di compilazione
- Il binding di $x.f$ è dinamico
 - L’identità di $x.f$ viene stabilita a tempo di esecuzione
- Il fatto che il binding di $x.f$ sia dinamico è una diretta conseguenza del polimorfismo e dell’overriding
- Ovvero, ciascun riferimento può puntare ad oggetti di tipo effettivo diverso (polimorfismo) e ciascuno di questi tipi effettivi può prevedere una versione diversa dello stesso metodo (overriding)
- Inoltre, il compilatore non può prevedere di che tipo effettivo sarà una variabile nel corso dell’esecuzione del programma (teoricamente, questo problema è *indecidibile*)

- In Java, il binding dei metodi (collegare ciascuna chiamata ad un metodo vero e proprio) avviene in due fasi:
 - **Early binding**, in cui il compilatore risolve l'*overloading* (scegliendo la firma più appropriata alla chiamata)
 - **Late binding**, in cui la JVM risolve l'*overriding* (scegliendo il metodo vero e proprio)
- Il late binding non è necessario per quei metodi che non ammettono overriding: i metodi **privati**, **statici** o **final**
- Per questi metodi si parla di “binding statico”, perché la scelta del metodo da eseguire viene fatta già dal compilatore

Come *non* viene scelta la firma più specifica

- Molti manuali di Java semplificano le regole dell'overloading, suggerendo che venga scelto il metodo che richiede *il minor numero di conversioni*
- Ad esempio, valutiamo l'invocazione `x.f(1, 2)`, supponendo che la classe di `x` offra i seguenti metodi:

```
public void f(double x, long y)
public void f(int x, double y)
```

Come *non* viene scelta la firma più specifica

- Molti manuali di Java semplificano le regole dell'overloading, suggerendo che venga scelto il metodo che richiede *il minor numero di conversioni*
- Ad esempio, valutiamo l'invocazione `x.f(1, 2)`, supponendo che la classe di `x` offra i seguenti metodi:

```
public void f(double x, long y)
public void f(int x, double y)
```

- Contiamo il numero di conversioni richieste, in due modi diversi:
 - Primo modo: contiamo quanti parametri richiedono una conversione
 - Secondo modo: contiamo il numero totale di “passi di conversione” richiesti (cioè, quanti archi dobbiamo percorrere nel grafo che rappresenta le conversioni implicite)
- Otteniamo i seguenti conteggi:

| | # parametri da convertire | # passi di conversione |
|--|---------------------------|------------------------|
| <code>public void f(double d, long l)</code> | 2 | 3 |
| <code>public void f(int d, double l)</code> | 1 | 2 |

- In entrambi i modi, sembra prevalere la seconda firma
- Invece, questo è un caso di **ambiguità** (errore di compilazione)
- Le prossime slide spiegano il perché

- L'early binding si divide a sua volta in due fasi:
 - 1) Individuazione delle firme candidate
 - 2) Scelta della firma più specifica tra quelle candidate
- Le prossime slide approfondiscono ciascuna di queste fasi

- Consideriamo una generica invocazione

$$x.f(a_1, \dots, a_n)$$

- Si ricorda che per “firma” di un metodo si intende il suo nome e l’elenco dei tipi dei suoi parametri formali
- Una generica firma

$$f(T_1, \dots, T_n)$$

è *candidata* per la chiamata in questione se:

- Si trova nella classe dichiarata di x o in una sua superclasse
 - E’ *visibile* dal punto della chiamata, rispetto alle regole di visibilità Java
 - E’ *compatibile* con la chiamata; ovvero, per ogni indice i compreso tra 1 ed n , il tipo (dichiarato) del parametro attuale a_i è **assegnabile** al tipo T_i
-
- Se nessuna firma risulta candidata per una data chiamata, il compilatore segnala un errore (accompagnato dal messaggio: “cannot find symbol”)

- Date due firme con lo stesso nome e numero di argomenti
 $f(T_1, \dots, T_n)$ e $f(U_1, \dots, U_n)$
- Si dice che la prima firma è **più specifica** della seconda se, per ogni indice i compreso tra 1 ed n, il tipo T_i è assegnabile al tipo U_i
- ATTENZIONE: notate che questo confronto tra firme non dipende dal tipo dei parametri attuali passati alla chiamata
- E' facile verificare che essere "più specifico" è una relazione riflessiva, antisimmetrica e transitiva, proprio come la relazione di assegnabilità
- Quindi, essa è una **relazione d'ordine** sull'insieme delle firme
- Tale ordine è **parziale**, in quanto alcune firme non sono confrontabili tra loro
- Ad esempio, le firme $f(int, double)$ e $f(double, int)$ non sono confrontabili quanto a specificità

- L'early binding si conclude individuando, tra le firme candidate, una che sia **più specifica di tutte le altre**
- Per simulare "a mano" questo meccanismo, nei casi complessi può essere conveniente realizzare un diagramma a **grafo**, in cui ci sia un nodo per ciascuna firma candidata ed un arco orientato da un nodo "a" ad un nodo "b" quando la firma "a" è più specifica della firma "b"
- Se nel grafo c'è un nodo che ha **archi uscenti diretti verso tutte le altre firme**, quella sarà la firma scelta dal compilatore
- Se nessuna firma è più specifica di tutte le altre, il compilatore segnala un errore e termina (parleremo di *chiamata ambigua*)
- Domanda: E' possibile che si trovi più di una firma più specifica di tutte le altre? Perché?
- Attenzione: in questa discussione sull'overloading sono state tralasciate la programmazione generica e l'autoboxing

Rivalutiamo l'invocazione `x.f(1, 2)`, supponendo che la classe di `x` offra i seguenti metodi:

```
public void f(double x, long y)
public void f(int x, double y)
```

- Il late binding è la fase di risoluzione dell'**overriding**, a carico della **JVM**
- Questa fase **riceve in input la firma** scelta dal compilatore durante l'early binding
- Consideriamo nuovamente la generica invocazione

x.f(a₁, ..., a_n)

La JVM cerca un metodo da eseguire, con il seguente algoritmo:

- si parte dalla classe effettiva di x
 - si cerca un metodo che abbia la firma **identica** a quella scelta dall'early binding
 - se non lo si trova, si passa alla superclasse
 - così via, fino ad arrivare ad Object
-
- Questo procedimento può fallire, cioè non trovare alcun metodo, solo in casi molto particolari
 - Ad esempio, se una classe A dipendeva da una classe B e la classe B è cambiata da quando è stata compilata A

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(double n, A x) { return "A1"; }  
    public String f(double n, B x) { return "A2"; }  
    public String f(int n, Object x) { return "A3"; }  
}  
  
class B extends A {  
    public String f(double n, B x) { return "B1"; }  
    public String f(float n, Object y) { return "B2"; }  
}  
  
class C extends A {  
    public final String f(int n, Object x) { return "C1"; }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = new B();  
        A alfa = beta;  
        System.out.println(alfa.f(3, beta));  
        System.out.println(alfa.f(3.0, beta));  
        System.out.println(beta.f(3.0, alfa));  
        System.out.println(gamma.f(3, gamma));  
        System.out.println(false ||  
                           alfa.equals(beta));  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso System.out.println), indicare la lista delle firme candidate

- Esaminiamo le chiamate una per volta

1) `System.out.println(alfa.f(3, beta));`

- alfa è di tipo dichiarato A, quindi le firme candidate vanno cercate nella classe A (o tutt'al più in Object)
- i due parametri attuali della chiamata sono di tipo (dichiarato) int e B, rispettivamente
- la firma `f(double, A)` è candidata, in quanto visibile e compatibile
 - essa è compatibile perché int è assegnabile a double (conversione implicita) e B è assegnabile ad A (sottotipo)
- la firma `f(double, B)` è candidata, in quanto visibile e compatibile
- la firma `f(int, Object)` è candidata, in quanto visibile e compatibile
- non vi sono altre firme candidate
- Delle tre firme candidate, la seconda è più specifica della prima, ma non è confrontabile con la terza
- Quindi, nessuna firma è più specifica di tutte le altre
- Il risultato è un **errore di compilazione**
- ATTENZIONE: ricordate che la scelta della firma più specifica non dipende dal tipo dei parametri attuali della chiamata

- Esaminiamo la seconda chiamata:

2) System.out.println(alfa.f(3.0, beta));

- alfa è di tipo dichiarato A, quindi le firme candidate vanno cercate nella classe A (o tutt'al più in Object)
- i due parametri attuali della chiamata sono di tipo (dichiarato) double e B, rispettivamente
- la firma f(double, A) è candidata, in quanto visibile e compatibile
- la firma f(double, B) è candidata, in quanto visibile e compatibile
- la firma f(int, Object) *non* è candidata, in quanto non compatibile
- non vi sono altre firme candidate
- Delle due firme candidate, la seconda è più specifica della prima
- Quindi, l'early binding si conclude con la selezione della firma f(double, B)
- Per il late binding, cerchiamo il metodo da eseguire a partire dalla classe effettiva di alfa: B
- Nella classe B, troviamo un metodo visibile con quella firma
- Quindi, l'output di questa chiamata è

B1

- Esaminiamo la terza chiamata:

3) System.out.println(beta.f(3.0, alfa));

- beta è di tipo dichiarato B, quindi le firme candidate vanno cercate in B, in A e in Object
- i due parametri attuali della chiamata sono di tipo (dichiarato) double ed A, rispettivamente
- la firma f(double, B) non è candidata, in quanto non compatibile (secondo argomento)
- la firma f(float, Object) non è candidata, in quanto non compatibile (primo argomento)
- la firma f(int, Object) non è candidata, in quanto non compatibile (primo argomento)
- la firma f(double, A) è candidata, in quanto visibile e compatibile
- Essendoci una sola firma candidata, l'early binding si conclude con la selezione della firma f(double, A)
- Per il late binding, cerchiamo il metodo da eseguire a partire dalla classe effettiva di beta: B
- Nella classe B, non c'è alcun metodo con la firma scelta
- Passiamo alla classe A, in cui troviamo un metodo con la firma scelta
- Quindi, l'output di questa chiamata è

A1

- Esaminiamo l'ultima chiamata:

4) System.out.println(gamma.f(3, gamma));

- gamma è di tipo dichiarato C, quindi le firme candidate vanno cercate in C, in A e in Object
- i due parametri attuali della chiamata sono di tipo (dichiarato) int e C, rispettivamente
- la firma f(int, Object) è candidata, in quanto visibile e compatibile
- la firma f(double, A) è candidata, in quanto visibile e compatibile
- la firma f(double, B) non è candidata, in quanto non compatibile (secondo argomento)
- Le due firme candidate non sono confrontabili
- Quindi, l'early binding si conclude con un **errore di compilazione**

Esercizio 2 (esame 27/11/2009, #2)

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(double n, Object x) { return "A1"; }  
    public String f(double n, A x) { return "A2"; }  
    public String f(int n, Object x) { return "A3"; }  
}  
class B extends A {  
    public String f(double n, Object x) { return "B1"; }  
    public String f(float n, Object y) { return "B2"; }  
}  
class C extends B {  
    public final String f(double n, A x) { return "C1"; }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = new B();  
        A alfa = gamma;  
        System.out.println(alfa.f(3.0, gamma));  
        System.out.println(beta.f(3, beta));  
        System.out.println(beta.f(3.0, null));  
        System.out.println(gamma.f(3.0, gamma));  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso System.out.println), indicare la lista delle firme candidate

Esercizio 3 (esame 25/1/2017, #1)

- Dato il seguente programma (tutte le classi appartengono allo stesso pacchetto):

```
class A {  
    public String f(A x, A[] y, B z) { return "A1"; }  
    public String f(A x, Object y, B z) { return "A2"; }  
}  
  
class B extends A {  
    public String f(B x, A[] y, B z) { return "B1:" + x.f((A)x, y, z); }  
    public String f(A x, B[] y, B z) { return "B2"; }  
}  
  
class C extends B {  
    public String f(A x, A[] y, C z) { return "C1:" + z.f(new C(), y, z); }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        C gamma = new C();  
        B beta = gamma;  
        A[] array = new A[10];  
        System.out.println(beta.f(gamma, array, gamma));  
        System.out.println(gamma.f(array[0], null, beta));  
        System.out.println(beta == gamma);  
    }  
}
```

- Indicare l'output del programma
- Se un'istruzione provoca un errore di compilazione, specificarlo e poi continuare l'esercizio ignorando quell'istruzione
- Per ogni chiamata ad un metodo (escluso System.out.println), indicare la lista delle firme candidate

Binding dinamico e auto-(un)boxing

- Nella risoluzione dell'overloading, l'autoboxing e l'auto-unboxing entrano in gioco *soltanto se necessario*
 - Ovvero, soltanto se altrimenti non ci sarebbero firme candidate
- Quindi, come **primo tentativo**, il compilatore cerca le firme che sono candidate senza prendere in considerazione l'autoboxing e l'auto-unboxing
- **Solo se non ci sono firme candidate**, il compilatore abilita le conversioni da tipo primitivo a tipo wrapper, e viceversa, e *riesamina tutte le firme (secondo tentativo)*
- Questa scelta è stata fatta per mantenere la compatibilità con il codice scritto prima dell'introduzione dell'auto-(un)boxing
- Infatti, le invocazioni a metodo che funzionavano senza auto-(un)boxing continuano a funzionare con l'auto-(un)boxing, e sono *risolte nello stesso modo*
- Con l'auto-(un)boxing, alcune invocazioni che prima non erano consentite diventano lecite

- Una volta ottenuto un insieme non vuoto di firme candidate, il compilatore passa alla scelta della più specifica, con le regole descritte nelle slide precedenti
- Quindi, l'auto-(un)boxing **non influenza** in alcun modo **la scelta della firma** più specifica
- Analogamente, l'auto-(un)boxing **non influenza** in alcun modo **il late binding**

- Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }
public static int foo(long i, String o) { return 2; }
```

- La chiamata

```
foo(new Integer(7), "ciao")
```

provoca un errore di **ambiguità**, perché il compilatore prima ottiene un insieme di firme candidate vuoto; poi, una volta attivato l'auto-(un)boxing, ottiene candidate **entrambe** le firme "foo", delle quali nessuna è più specifica dell'altra

- Consideriamo i seguenti metodi:

```
public static int foo(int i, Object o) { return 1; }
public static int foo(long i, String o) { return 2; }
```

- La chiamata `foo(new Float(7), "ciao")` provoca un **errore** di compilazione, in quanto il compilatore non trova firme candidate neanche al secondo tentativo
- La chiamata `foo(new Long(7), "ciao")` ottiene output 2, in quanto quella è l'unica firma candidata, una volta attivato l'auto-(un)boxing

- Consideriamo i seguenti metodi:

```
public static int bar(double a, Integer b) { return 3; }
public static int bar(Double a, Integer b) { return 4; }
```

- La chiamata `bar(1.0, 7)` provoca un errore di ambiguità, perché entrambe le firme saranno candidate (al secondo tentativo)
- La chiamata `bar(1, 7)` ottiene il risultato 3, perché avrà un'unica firma candidata (al secondo tentativo)
 - la seconda firma non è candidata perché un `int` non può trasformarsi in `Double` tramite autoboxing

Linguaggi di Programmazione I – Lezione 2

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

29 marzo 2022



Panoramica della lezione

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Modello imperativo



Memoria

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

■ Consiste in un insieme di “contenitori di dati” ...

- ◆ Ad es. parole (o celle) della memoria centrale
- ◆ Tipicamente rappresentate dal loro indirizzo



Memoria

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente
Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

■ Consiste in un insieme di “contenitori di dati” ...

- ◆ Ad es. parole (o celle) della memoria centrale
- ◆ Tipicamente rappresentate dal loro indirizzo

■ ... associati ai valori in essi contenuti

- ◆ I valori delle variabili



Memoria

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente
Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- Consiste in un insieme di “contenitori di dati” ...
 - ◆ Ad es. parole (o celle) della memoria centrale
 - ◆ Tipicamente rappresentate dal loro indirizzo
- ... associati ai valori in essi contenuti
 - ◆ I valori delle variabili
- Dunque (concettualmente) la memoria è
 - ◆ Una funzione da uno spazio di locazioni ad uno spazio di valori
 - ◆ $\text{mem} : \text{Indirizzi} \rightarrow \text{Valori}$
 - ◆ **mem(loc)** = “valore contenuto nella locazione loc”



Grammatica in forma Backus-Naur

- Una sintassi per le grammatiche context-free
- Invece di “ $\text{exp} \rightarrow \text{exp} + \text{exp}$ ” ...
- ...scrivereemo “ $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle$ ”

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Grammatica in forma Backus-Naur

[Modello imperativo](#)

[Memoria](#)

[Grammatica in
forma Backus-Naur](#)

[Assegnazioni](#)

[Modello imperativo](#)

[Ambiente](#)

[Esempi di ambiente:
imperativo](#)

[Esempi di ambiente:
funzionale](#)

[Esempio:
assegnazione](#)

[Data Object e
legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di locazione](#)

- Una sintassi per le grammatiche context-free
- Invece di “ $\text{exp} \rightarrow \text{exp} + \text{exp}$ ”...
- ...scrivereemo “ $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle$ ”

Si ricordi che:

- “ exp ” è un *simbolo non-terminale*
- “ $+$ ” è un *simbolo terminale*
- “ $\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle$ ” è una *produzione o regola*



Assegnazioni

- Definizione grammaticale (sintassi):

```
<assegnazione> ::= <name> <assignment-operator> <expression>
```

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.



Assegnazioni

- Definizione grammaticale (sintassi):

```
<assegnazione> ::= <name> <assignment-operator> <expression>
```

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Esempio in Pascal:

```
a := b + c;
```



Assegnazioni

■ Definizione grammaticale (sintassi):

```
<assegnazione> ::= <name> <assignment-operator> <expression>
```

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Esempio in Pascal:

```
a := b + c;
```

■ Esecuzione (significato, semantica):

Il valore di <expression> va memorizzato nell'indirizzo rappresentato da <name>.



Assegnazioni

■ Definizione grammaticale (sintassi):

```
<assegnazione> ::= <name> <assignment-operator> <expression>
```

<name> rappresenta la locazione dove viene posto il risultato mentre in <expression> sono specificati una computazione e i riferimenti ai valori necessari alla computazione.

Esempio in Pascal:

```
a := b + c;
```

■ Esecuzione (significato, semantica):

Il valore di <expression> va memorizzato nell'indirizzo rappresentato da <name>.

Il valore di <expression> dipenderà dai valori contenuti negli indirizzi degli argomenti di <expression> rappresentati dai nomi di questi ottenuto seguendo le prescrizioni del codice associato al suo nome.



Modello imperativo

- È il più vicino agli elaboratori “standard”.

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Modello imperativo

[Modello imperativo](#)

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di locazione](#)

- È il più vicino agli elaboratori “standard”.
- I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.
 - ◆ è anche il modo di pilotare i display, salvare su memorie periferiche ecc.



Modello imperativo

[Modello imperativo](#)

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:

assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- È il più vicino agli elaboratori “standard”.
- I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.
 - ◆ è anche il modo di pilotare i display, salvare su memorie periferiche ecc.
- Ogni unità di esecuzione consiste di 4 passi:
 1. ottenere indirizzi delle locazioni di operandi e risultato;



Modello imperativo

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- È il più vicino agli elaboratori “standard”.
- I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.
 - ◆ è anche il modo di pilotare i display, salvare su memorie periferiche ecc.
- Ogni unità di esecuzione consiste di 4 passi:
 1. ottenere indirizzi delle locazioni di operandi e risultato;
 2. ottenere dati di operandi da locazioni di operandi;



Modello imperativo

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:

assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- È il più vicino agli elaboratori “standard”.
- I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.
 - ◆ è anche il modo di pilotare i display, salvare su memorie periferiche ecc.
- Ogni unità di esecuzione consiste di 4 passi:
 1. ottenere indirizzi delle locazioni di operandi e risultato;
 2. ottenere dati di operandi da locazioni di operandi;
 3. valutare risultato;



Modello imperativo

Modello imperativo

Memoria
Grammatica in
forma Backus-Naur
Assegnazioni

Modello imperativo

Ambiente
Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- È il più vicino agli elaboratori “standard”.
- I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.
 - ◆ è anche il modo di pilotare i display, salvare su memorie periferiche ecc.
- Ogni unità di esecuzione consiste di 4 passi:
 1. ottenere indirizzi delle locazioni di operandi e risultato;
 2. ottenere dati di operandi da locazioni di operandi;
 3. valutare risultato;
 4. memorizzare risultato in locazione risultato.

In sostanza un assegnamento!



Modello imperativo

Modello imperativo

Memoria
Grammatica in
forma Backus-Naur
Assegnazioni

Modello imperativo

Ambiente
Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- È il più vicino agli elaboratori “standard”.
- I programmi sono descrizioni di sequenze di modifiche della “memoria” del calcolatore.
 - ◆ è anche il modo di pilotare i display, salvare su memorie periferiche ecc.
- Ogni unità di esecuzione consiste di 4 passi:
 1. ottenere indirizzi delle locazioni di operandi e risultato;
 2. ottenere dati di operandi da locazioni di operandi;
 3. valutare risultato;
 4. memorizzare risultato in locazione risultato.

In sostanza un assegnamento!

- Il modello imperativo usa dei nomi come astrazione di indirizzi di locazioni di memoria (variabili).



Ambiente (di esecuzione)

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

L'*ambiente* (environment) comprende:

- Un **insieme di nomi** (di variabili, parametri, funzioni, etc.)
- ... associati a qualcosa da cui si può risalire al **valore** della variabile o del parametro.



Ambiente (di esecuzione)

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

L'ambiente (environment) comprende:

- Un **insieme di nomi** (di variabili, parametri, funzioni, etc.)
- ... associati a qualcosa da cui si può risalire al **valore** della variabile o del parametro.
- Concettualmente l'ambiente è: una funzione da un insieme di identificatori (i nomi) a un insieme di ...?
 $\text{env(id)} = ???$



Ambiente (di esecuzione)

Modello imperativo

Memoria

Grammatica in forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente: imperativo

Esempi di ambiente: funzionale

Esempio: assegnazione

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

L'ambiente (environment) comprende:

- Un **insieme di nomi** (di variabili, parametri, funzioni, etc.)
- ... associati a qualcosa da cui si può risalire al **valore** della variabile o del parametro.
- Concettualmente l'ambiente è: una funzione da un insieme di identificatori (i nomi) a un insieme di ...?
 $\text{env(id)} = ???$
- Il codominio della funzione dipende dal paradigma computazionale del linguaggio.



Esempi di ambiente: imperativo

- Nel paradigma imperativo, la funzione env associa gli identificatori a locazioni di memoria, le quali, a loro volta, sono associate (funzione mem) al contenuto di memoria
- env: Nomi → Indirizzi
- **env(x) = indirizzo della variabile di nome x**
- Il valore di una variabile x è mem(env(x))

Modello imperativo

Memoria
Grammatica in
forma Backus-Naur
Assegnazioni

Modello imperativo

Ambiente
**Esempi di ambiente:
imperativo**

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Esempi di ambiente: imperativo

Modello imperativo

Memoria
Grammatica in
forma Backus-Naur
Assegnazioni

Modello imperativo

Ambiente
**Esempi di ambiente:
imperativo**

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

- Nel paradigma imperativo, la funzione env associa gli identificatori a locazioni di memoria, le quali, a loro volta, sono associate (funzione mem) al contenuto di memoria
- env: Nomi → Indirizzi
- **env(x) = indirizzo della variabile di nome x**
- Il valore di una variabile x è mem(env(x))
- Attenzione: env(x) è immutabile finchè x esiste...
 - ◆ vedremo che gli identificatori nascono e muoiono durante l'esecuzione

nel paradigma imperativo la funzione env identifica una associazione *immutabile* (la locazione di memoria associata a un nome non cambia);



Esempi di ambiente: funzionale

- Nel paradigma funzionale (puro), il valore associato a un nome non cambia nel tempo
- Quindi, la funzione env associa direttamente gli identificatori al contenuto della memoria
- env: Nomi → Valori
- Il valore di una variabile x è env(x)

Modello imperativo

Memoria

Grammatica in forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Esempi di ambiente: funzionale

- Nel paradigma funzionale (puro), il valore associato a un nome non cambia nel tempo
- Quindi, la funzione env associa direttamente gli identificatori al contenuto della memoria
- env: Nomi → Valori
- Il valore di una variabile x è $\text{env}(x)$
- Anche in questo caso, env è una associazione immutabile (fintanto che la variabile x è in scope)

Modello imperativo

Memoria
Grammatica in
forma Backus-Naur
Assegnazioni

Modello imperativo

Ambiente
Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Esempi di ambiente: funzionale

- Nel paradigma funzionale (puro), il valore associato a un nome non cambia nel tempo
- Quindi, la funzione env associa direttamente gli identificatori al contenuto della memoria
- env: Nomi → Valori
- Il valore di una variabile x è env(x)
- Anche in questo caso, env è una associazione immutabile (fintanto che la variabile x è in scope)
- Nel paradigma funzionale non esiste la funzione mem

Modello imperativo

Memoria
Grammatica in
forma Backus-Naur
Assegnazioni

Modello imperativo

Ambiente
Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale
Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Esempio: assegnazione

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

In una assegnazione:

```
x := x + 1;
```

il valore dell'espressione a destra va salvato nella variabile a sinistra
quindi:



Esempio: assegnazione

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

In una assegnazione:

```
x := x + 1;
```

il valore dell'espressione a destra va salvato nella variabile a sinistra
quindi:

la x di sinistra indica la locazione associata al nome (cioè $\text{env}(x)$)



Esempio: assegnazione

Modello imperativo

Memoria

Grammatica in
forma Backus-Naur

Assegnazioni

Modello imperativo

Ambiente

Esempi di ambiente:
imperativo

Esempi di ambiente:
funzionale

Esempio:
assegnazione

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

In una assegnazione:

```
x := x + 1;
```

il valore dell'espressione a destra va salvato nella variabile a sinistra
quindi:

la x di sinistra indica la locazione associata al nome (cioè $\text{env}(x)$)

la x di destra indica il valore della variabile (cioè $\text{mem}(\text{env}(x))$)



Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Data Object e legami



Data Object: un altro modo di astrarre l'implementazione dei dati

- Ogni variabile, parametro, oggetto, ... può essere rappresentato con un *data object*
- Un *data object* è una quadrupla (L, N, V, T) , ove:
 - ◆ L : locazione.
 - ◆ N : nome.
 - ◆ V : valore.
 - ◆ T : tipo.

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Data Object: un altro modo di astrarre l'implementazione dei dati

- Ogni variabile, parametro, oggetto, ... può essere rappresentato con un *data object*
- Un *data object* è una quadrupla (L, N, V, T) , ove:
 - ◆ L : locazione.
 - ◆ N : nome.
 - ◆ V : valore.
 - ◆ T : tipo.
- Un *legame* è la determinazione di una delle componenti.
 - ◆ ad esempio legame di locazione, di nome, di valore, di tipo

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Data Object: un altro modo di astrarre

l'implementazione dei dati

- Ogni variabile, parametro, oggetto, ... può essere rappresentato con un *data object*
- Un *data object* è una quadrupla (L, N, V, T) , ove:
 - ◆ L : locazione.
 - ◆ N : nome.
 - ◆ V : valore.
 - ◆ T : tipo.
- Un *legame* è la determinazione di una delle componenti.
 - ◆ ad esempio legame di locazione, di nome, di valore, di tipo
- Nel paradigma imperativo
 - ◆ il legame di locazione corrisponde alla funzione *env*
 - ◆ il legame di valore di una variabile di nome x corrisponde a $mem(env(x))$

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

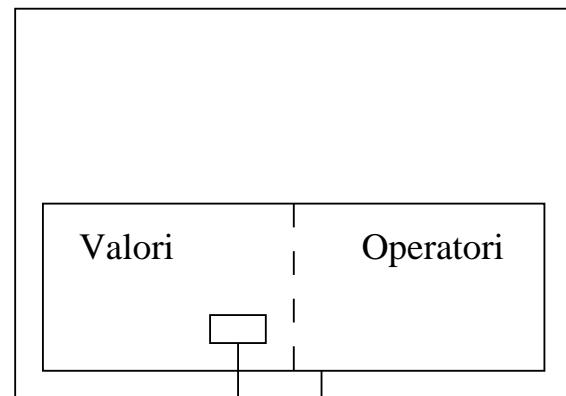
Data object in C

Legami di tipo

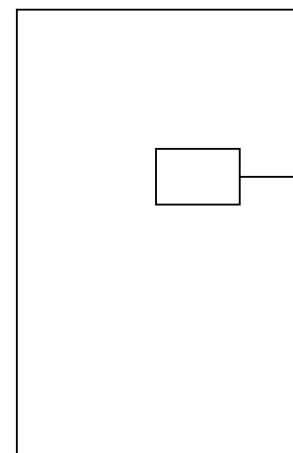
Blocchi di istruzioni

Legami di locazione

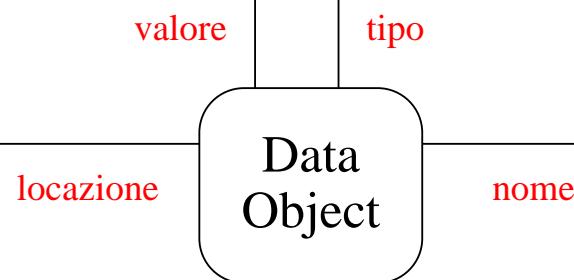
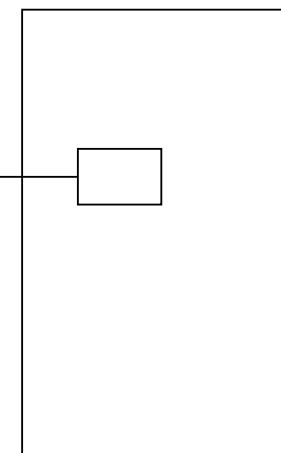
Spazio di tipi



Spazio di memoria



Spazio di nomi





Modifiche di legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).



Modifiche di legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).



Modifiche di legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).



Modifiche di legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).

- il *location binding* avviene durante il caricamento in memoria, oppure a run-time (si veda la gestione dei blocchi più avanti);



Modifiche di legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).

- il *location binding* avviene durante il caricamento in memoria, oppure a run-time (si veda la gestione dei blocchi più avanti);
- il *name binding* avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione;



Modifiche di legami

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Variazioni di legami (binding) possono avvenire:

1. Durante la compilazione (compile time).
2. Durante il caricamento in memoria (load time).
3. Durante l'esecuzione (run time).

- il *location binding* avviene durante il caricamento in memoria, oppure a run-time (si veda la gestione dei blocchi più avanti);
- il *name binding* avviene durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione;
- il *type binding* avviene (di solito, si veda dopo) durante la compilazione, nell'istante in cui il compilatore incontra una dichiarazione di tipo; un tipo è definito dal sottospazio di valori (e dai relativi operatori) che un *data object* può assumere.



Esempio 1

Modello imperativo

Data Object e legami

Data Object: un altro modo di astrarre l'implementazione dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

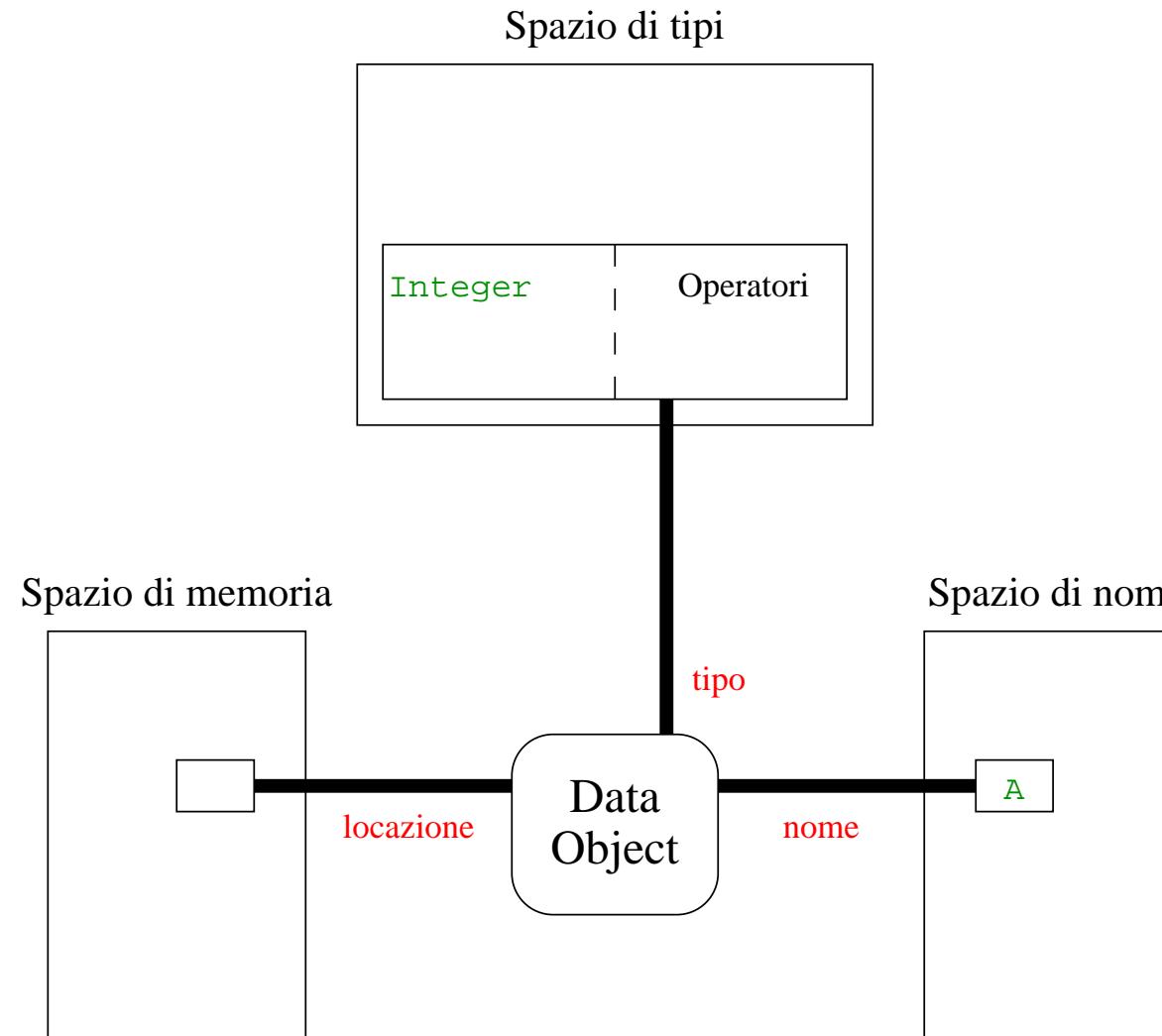
Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



A: integer;



Esempio 2

Modello imperativo

Data Object e legami

Data Object: un altro modo di astrarre l'implementazione dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

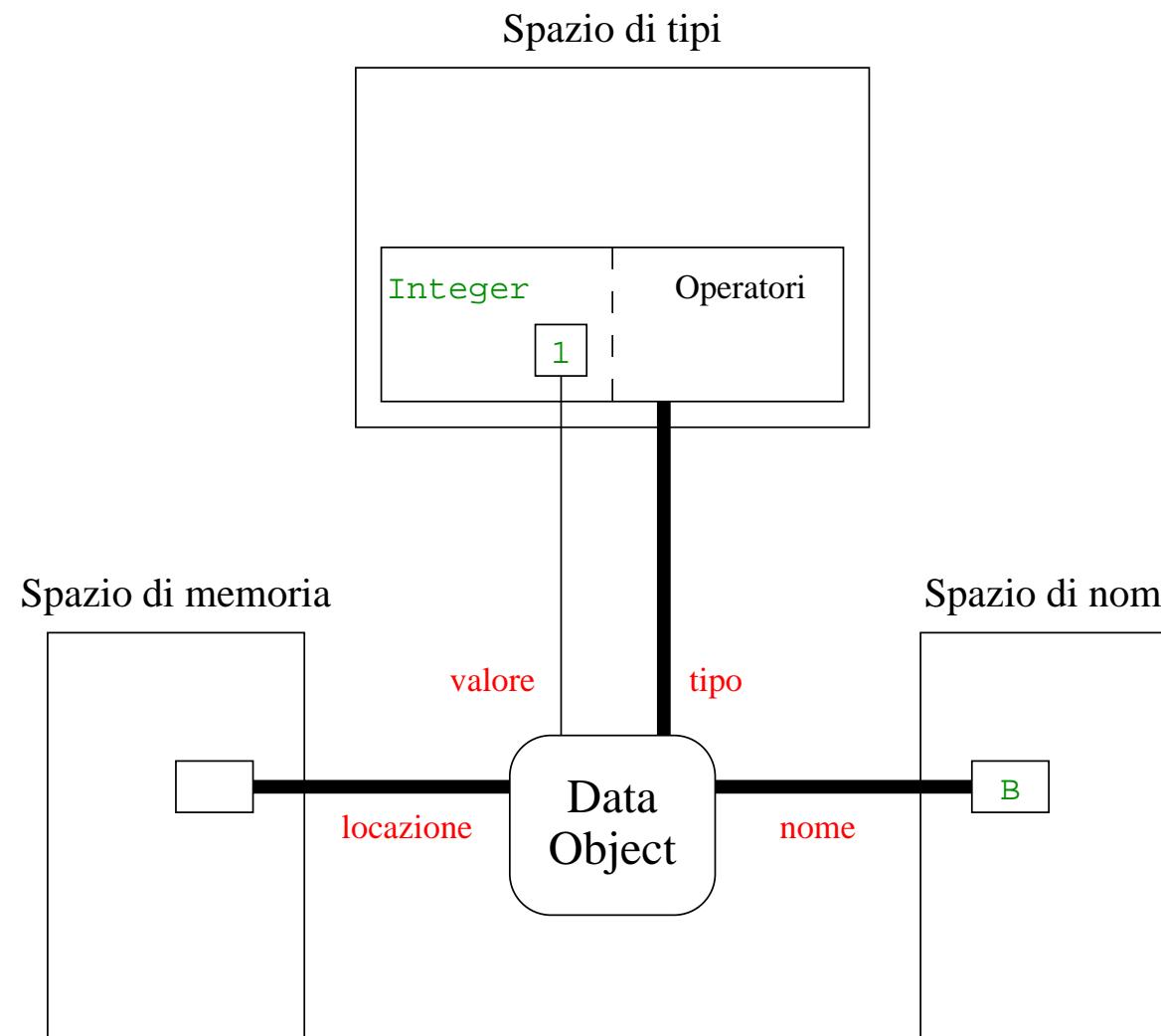
Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



B: integer:= 1;



Esempio 3

Modello imperativo

Data Object e legami

Data Object: un altro modo di astrarre l'implementazione dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

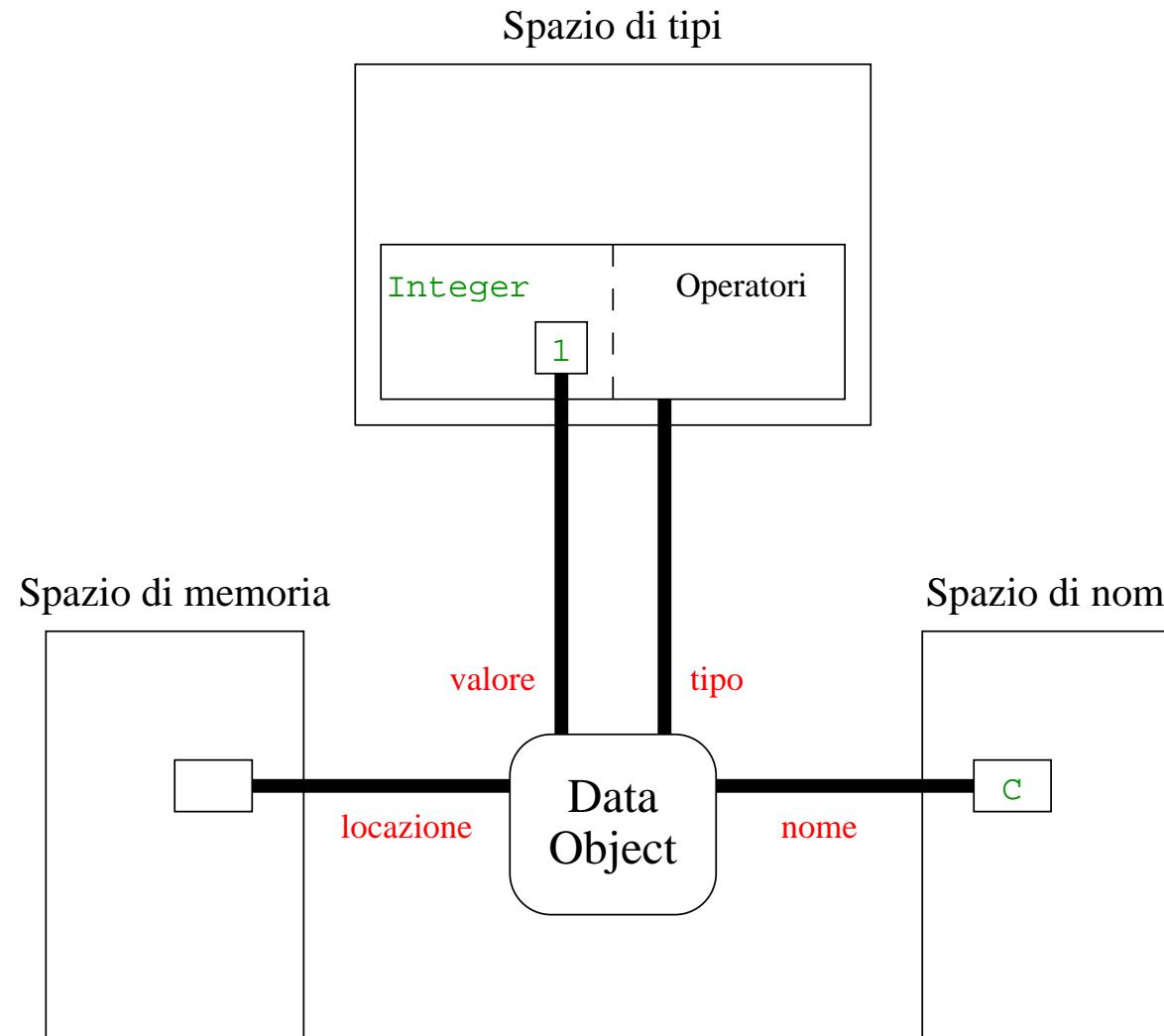
Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



C: constant integer:= 1;



Il puntatore (1)

Modello imperativo

Data Object e legami

Data Object: un altro modo di astrarre l'implementazione dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

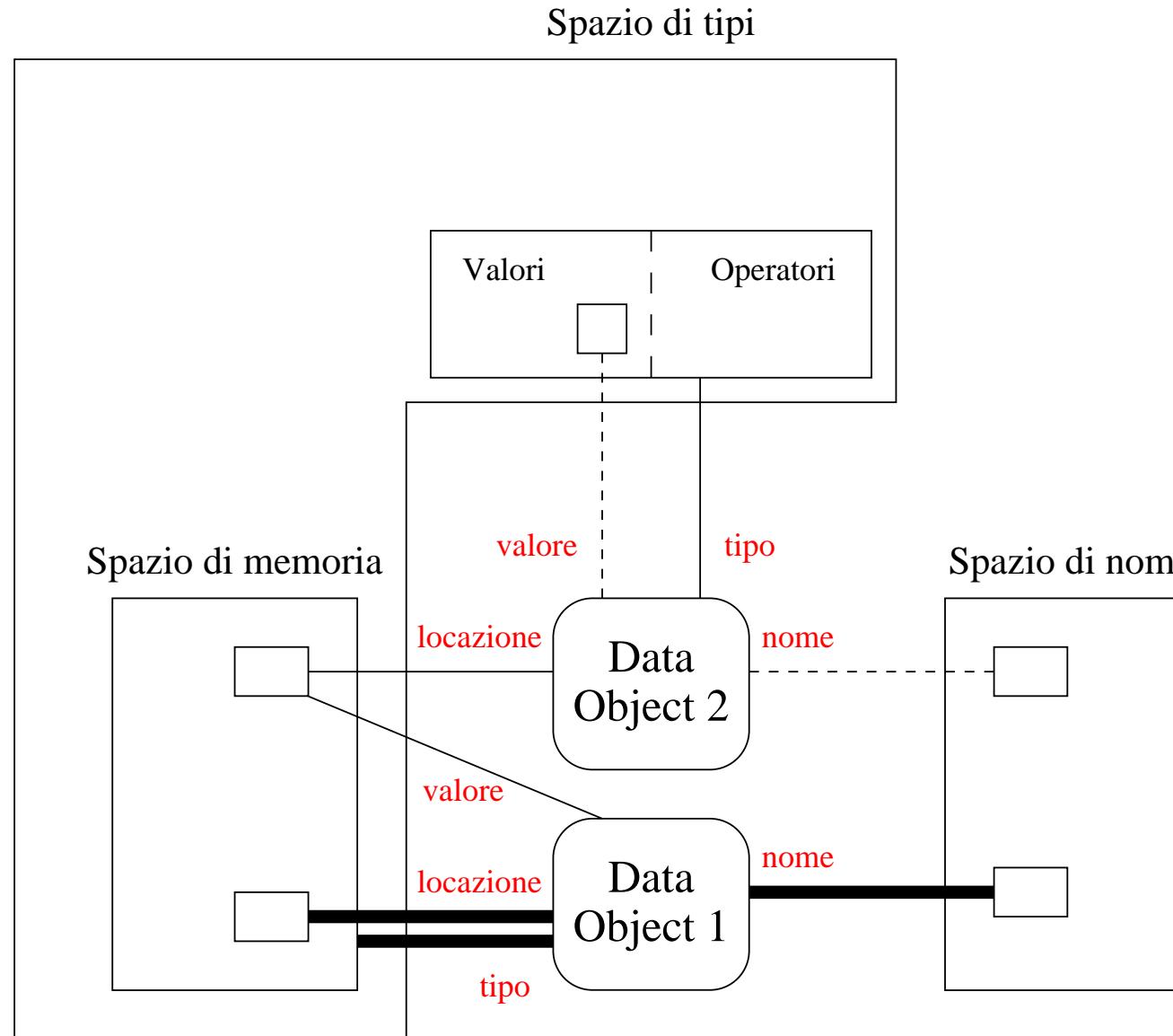
Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione





Il puntatore (2)

- 2 *data object* coinvolti.

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Il puntatore (2)

- 2 *data object* coinvolti.
- Il secondo può non avere un legame di nome o di valore.

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Il puntatore (2)

- 2 *data object* coinvolti.
- Il secondo può non avere un legame di nome o di valore.
- La deallocazione è necessaria, perché la modifica del legame di valore genera di solito dati non più accessibili per nome o riferimento.

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Il puntatore (2)

- 2 *data object* coinvolti.
- Il secondo può non avere un legame di nome o di valore.
- La deallocazione è necessaria, perché la modifica del legame di valore genera di solito dati non più accessibili per nome o riferimento.
- Alcuni linguaggi possiedono meccanismi di recupero automatico di memoria (*garbage collector*).

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione



Data object in C

Modello imperativo

Data Object e
legami

Data Object: un
altro modo di
astrarre
l'implementazione
dei dati

Legami

Modifiche di legami

Esempio 1

Esempio 2

Esempio 3

Il puntatore (1)

Il puntatore (2)

Data object in C

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Inserire slide “Richiami di C”



Modello imperativo

Data Object e
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

Legami di tipo



Legame di tipo

- Per definizione è correlato al legame di valore (ad es. il tipo di una variabile e il tipo del suo valore devono corrispondere).



Legame di tipo

- Per definizione è correlato al legame di valore (ad es. il tipo di una variabile e il tipo del suo valore devono corrispondere).
- Ogni volta che il legame di valore viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di tipo.



Legame di tipo

- Per definizione è correlato al legame di valore (ad es. il tipo di una variabile e il tipo del suo valore devono corrispondere).
- Ogni volta che il legame di valore viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di tipo.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.



Legame di tipo

- Per definizione è correlato al legame di valore (ad es. il tipo di una variabile e il tipo del suo valore devono corrispondere).
- Ogni volta che il legame di valore viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di tipo.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.

Esempio: nei linguaggi di scripting e Python

```
x=1; ... x= "abc";
```

Inizialmente il tipo di x è numerico, poi è stringa (il legame di tipo cambia in seguito ad un cambio del legame di valore).



Legame di tipo

- Per definizione è correlato al legame di valore (ad es. il tipo di una variabile e il tipo del suo valore devono corrispondere).
- Ogni volta che il legame di valore viene modificato, occorrerebbe controllare (type checking) la consistenza con il legame di tipo.
- Un linguaggio è *dinamicamente tipizzato* se il legame (e le variazioni di legame) e di conseguenza anche il controllo di consistenza (se avviene) avvengono durante l'esecuzione.

Esempio: nei linguaggi di scripting e Python

```
x=1; ... x= "abc";
```

Inizialmente il tipo di x è numerico, poi è stringa (il legame di tipo cambia in seguito ad un cambio del legame di valore).

- Un linguaggio è *staticamente tipizzato* se il legame avviene durante la compilazione; in questo caso il controllo di consistenza (se avviene) può avvenire in entrambe le fasi.



Type checking (1)

Modello imperativo

Data Object e
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.



Type checking (1)

Modello imperativo

Data Object e
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.



Type checking (1)

Modello imperativo

Data Object e
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre



Type checking (1)

Modello imperativo

Data Object e
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre
 - ◆ Java è fortemente tipizzato (vedremo poi).



Type checking (1)

Modello imperativo

Data Object e
legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre
approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

È il meccanismo di controllo di consistenza della coppia dei legami valore-tipo.

Può avvenire: a) durante la compilazione, b) durante l'esecuzione, c) per nulla.

- Un linguaggio è *fortemente tipizzato* se il controllo di consistenza avviene sempre
 - ◆ Java è fortemente tipizzato (vedremo poi).
 - ◆ Pascal è quasi fortemente tipizzato (una sola eccezione di assenza di controllo: i record con varianti).



Type checking (2)

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione



Type checking (2)

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.
 - ◆ C è debolmente tipizzato:
 - (1) supporta le operazioni di *casting* (o *coercizione di tipo*), che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo;

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso sbagliato di union non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del controllo di tipi

Blocchi di istruzioni

Legami di locazione



Type checking (2)

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.
 - ◆ C è debolmente tipizzato:
 - (1) supporta le operazioni di *casting* (o *coercizione di tipo*), che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo;
 - (2) supporta conversioni implicite (senza cast) tra tutti i tipi di puntatori;

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso sbagliato di union non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del controllo di tipi

Blocchi di istruzioni

Legami di locazione



Type checking (2)

- Un linguaggio è *debolmente tipizzato* se il controllo di consistenza può non avvenire affatto in numerosi casi.
 - ◆ C è debolmente tipizzato:
 - (1) supporta le operazioni di *casting* (o *coercizione di tipo*), che consentono di forzare, in esecuzione, l'interpretazione di un qualunque valore secondo un qualunque tipo;
 - (2) supporta conversioni implicite (senza cast) tra tutti i tipi di puntatori;
 - (3) supporta le *unioni*, che sovrappongono la locazione di variabili di tipo diverso, senza controllare se il valore corrisponde al tipo con cui viene usato

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso sbagliato di union non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del controllo di tipi

Blocchi di istruzioni

Legami di locazione



Esempio di uso sbagliato di union non segnalato

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso sbagliato di union non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

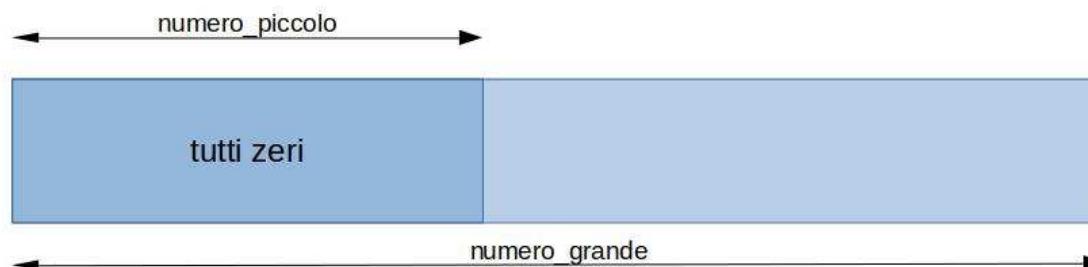
approssimazioni del controllo di tipi

Blocchi di istruzioni

Legami di locazione

```
union numero
{
    int numero_piccolo;
    double numero_grande;
} numeri;

void main() {
    numeri.numero_grande = 3.0;
    printf("%d\n", numeri.numero_piccolo);
}
```



- si inserisce un double ma si legge come fosse un int
- viene stampato '0'
- il compilatore non segnala l'errore



Il linguaggio perfetto (1)

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso sbagliato di union non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del controllo di tipi

Blocchi di istruzioni

Legami di locazione

Sarebbe bello se esistesse un linguaggio Turing completo, in cui il type checking avvenisse completamente durante la compilazione e *in cui il compilatore non generasse più errori del necessario* – Linguaggio Perfetto (LP).

- LP, se esistesse, sarebbe staticamente e fortemente tipizzato (il più “forte” di tutti i fortemente tipizzati).



Il linguaggio perfetto (1)

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso
sbagliato di union
non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del
controllo di tipi

Blocchi di istruzioni

Legami di locazione

Sarebbe bello se esistesse un linguaggio Turing completo, in cui il type checking avvenisse completamente durante la compilazione e *in cui il compilatore non generasse più errori del necessario* – Linguaggio Perfetto (LP).

- LP, se esistesse, sarebbe staticamente e fortemente tipizzato (il più “forte” di tutti i fortemente tipizzati).
- LP non può esistere.



Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

```
int x;  
P;  
x = "pippo";
```

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P (l'istruzione errata viene eseguita se e solo se la chiamata a P termina)

quindi LP non sarebbe Turing completo, contro l'ipotesi.



Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

```
int x;  
P;  
x = "pippo";
```

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P (l'istruzione errata viene eseguita se e solo se la chiamata a P termina)

quindi LP non sarebbe Turing completo, contro l'ipotesi.

- I compilatori, in situazioni come la precedente, ignorano se una istruzione viene eseguita o meno e segnalano comunque un errore nell'ultima linea.



Il linguaggio perfetto (2)

- Se LP esistesse, allora il compilatore, per essere capace di decidere la correttezza dell'ultima assegnazione in:

```
int x;  
P;  
x = "pippo";
```

dove P è un generico programma, dovrebbe essere capace di decidere la terminazione di P (l'istruzione errata viene eseguita se e solo se la chiamata a P termina)

quindi LP non sarebbe Turing completo, contro l'ipotesi.

- I compilatori, in situazioni come la precedente, ignorano se una istruzione viene eseguita o meno e segnalano comunque un errore nell'ultima linea.

Per esempio, questo programma Pascal non compila anche se l'istruzione errata non verrebbe eseguita:

```
program wrong (input, output);  
var i: integer;  
begin  
  if false then i := 3.14;  
    else i := 0;  
end.
```



Altre approssimazioni del controllo di tipi

- Alcuni controlli di tipo sono impossibili a tempo di compilazione
 - ◆ ad esempio la correttezza dei “down cast” in Java, che vedremo più avanti
- Quindi la strategia per i linguaggi fortemente e staticamente tipati è
 - ◆ fare quanti più controlli possibile a tempo di compilazione
 - ◆ eseguire i rimanenti a tempo di esecuzioneperchè prima si scoprono gli errori e più il testing diventa economico

Modello imperativo

Data Object e legami

Legami di tipo

Legame di tipo

Type checking (1)

Type checking (2)

Esempio di uso sbagliato di union non segnalato

Linguaggio perfetto

Linguaggio perfetto

Altre

approssimazioni del controllo di tipi

Blocchi di istruzioni

Legami di locazione



Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

Blocchi di istruzioni



Scope

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

Si definisce *scope* o *ambito di validità* di un legame, l'insieme degli enunciati del programma in cui quel legame è valido.

I linguaggi moderni limitano lo scope usando *blocchi* di codice

I blocchi possono anche prendere la forma di procedure, funzioni, classi, etc.



Necessità dei blocchi

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping statico

Mascheramento

Mascheramento

Legami di locazione

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo (*Dove finisce il corpo di questo ciclo?*)



Necessità dei blocchi

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni
Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping statico

Mascheramento

Mascheramento

Legami di locazione

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo (*Dove finisce il corpo di questo ciclo?*)
- ambito di una procedura (*Dove finisce questa procedura?*)



Necessità dei blocchi

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping statico

Mascheramento

Mascheramento

Legami di locazione

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo (*Dove finisce il corpo di questo ciclo?*)
- ambito di una procedura (*Dove finisce questa procedura?*)
- unità di compilazione separata;



Necessità dei blocchi

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping statico

Mascheramento

Mascheramento

Legami di locazione

Istruzioni raggruppate in blocchi per meglio definire:

- ambito delle strutture di controllo (*Dove finisce il corpo di questo ciclo?*)
- ambito di una procedura (*Dove finisce questa procedura?*)
- unità di compilazione separata;
- ambito di validità dei legami di nome (*Dove è visibile questo nome?*)



Definizioni

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

A scopo didattico, introduciamo un semplice pseudo-linguaggio di *blocchi di codice*.

Un blocco contiene due parti:

- una sezione di dichiarazione di nomi (cioè, variabili);



Definizioni

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

A scopo didattico, introduciamo un semplice pseudo-linguaggio di *blocchi di codice*.

Un blocco contiene due parti:

- una sezione di dichiarazione di nomi (cioè, variabili);
- una sezione che comprende gli enunciati sui quali hanno validità quei legami di nome.



Definizioni

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

A scopo didattico, introduciamo un semplice pseudo-linguaggio di *blocchi di codice*.

Un blocco contiene due parti:

- una sezione di dichiarazione di nomi (cioè, variabili);
- una sezione che comprende gli enunciati sui quali hanno validità quei legami di nome.

Esempio:

```
...
BLOCK A;
    DECLARE I;
BEGIN A
    ...
        {I DEL BLOCCO A}
END A;
...
```



Ambito di validità di legami

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

Essenzialmente due tipi di ambito di validità (scoping):



Ambito di validità di legami

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni
Scope

Necessità dei blocchi
Definizioni

Ambito di . . .
Esempio di scoping
statico

Mascheramento
Mascheramento

Legami di locazione

Essenzialmente due tipi di ambito di validità (scoping):

Scoping statico o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporne di nuovi.



Ambito di validità di legami

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni
Scope

Necessità dei blocchi
Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento
Mascheramento

Legami di locazione

Essenzialmente due tipi di ambito di validità (scoping):

Scoping statico o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporne di nuovi.

Scoping dinamico

Concetto qui esaminato solo in relazione ai blocchi annidati, ma che assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate. In questo caso la procedura chiamata vede e usa i legami visti e usati dalla procedura chiamante.



Ambito di validità di legami

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni
Scope

Necessità dei blocchi
Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento
Mascheramento

Legami di locazione

Essenzialmente due tipi di ambito di validità (scoping):

Scoping statico o lessicale.

Blocchi annidati vedono e usano i legami dei blocchi più esterni (*legami non locali*) e, di solito, possono aggiungere legami locali o sovrapporne di nuovi.

Scoping dinamico

Concetto qui esaminato solo in relazione ai blocchi annidati, ma che assume il proprio senso maggiore quando vi sono procedure chiamanti e chiamate. In questo caso la procedura chiamata vede e usa i legami visti e usati dalla procedura chiamante.

Esamineremo tutti i dettagli nella prossima lezione.



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;  
    DECLARE X;  
BEGIN P  
    ...           {X da P}
```



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;  
    DECLARE X;  
BEGIN P  
    ...           {X da P}  
    BLOCK A;  
        DECLARE Y;  
    BEGIN A  
        ...           {X da P, Y da A}
```



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;
    DECLARE X;
BEGIN P
    ...
    BLOCK A;
        DECLARE Y;
    BEGIN A
        ...
        BLOCK B;
            DECLARE Z;
    BEGIN B
        ...
    {X da P}                                {X da P, Y da A}
    {X da P, Y da A, Z da B}
```



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;
    DECLARE X;
BEGIN P
    ...
    BLOCK A;
        DECLARE Y;
    BEGIN A
        ...
        {X da P, Y da A}
    BLOCK B;
        DECLARE Z;
    BEGIN B
        ...
        {X da P, Y da A, Z da B}
    END B;
    ...
    {X da P, Y da A}
```



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;
    DECLARE X;
BEGIN P
    ...
    BLOCK A;
        DECLARE Y;
    BEGIN A
        ...
        {X da P, Y da A}
    BLOCK B;
        DECLARE Z;
    BEGIN B
        ...
        {X da P, Y da A, Z da B}
    END B;
    ...
    {X da P, Y da A}
END A;
...
{X da P}
```



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;
    DECLARE X;
BEGIN P
    ...
    BLOCK A;
        DECLARE Y;
    BEGIN A
        ...
        {X da P, Y da A}
        BLOCK B;
            DECLARE Z;
    BEGIN B
        ...
        {X da P, Y da A, Z da B}
    END B;
    ...
    {X da P, Y da A}
END A;
...
{X da P}
BLOCK C;
    DECLARE Z;
START C
...
{X da P, Z da C}
```



Esempio di scoping statico

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

**Esempio di scoping
statico**

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;
    DECLARE X;
BEGIN P
    ...
    BLOCK A;
        DECLARE Y;
    BEGIN A
        ...
        {X da P, Y da A}
        BLOCK B;
            DECLARE Z;
    BEGIN B
        ...
        {X da P, Y da A, Z da B}
    END B;
    ...
    {X da P, Y da A}
END A;
...
{X da P}
BLOCK C;
    DECLARE Z;
START C
    ...
    {X da P, Z da C}
END C;
...
{X da P}
END P;
```



Mascheramento

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

Esempio di scoping statico

Mascheramento

Mascheramento

Legami di locazione

- Si parla di *mascheramento* o *shadowing* quando un blocco interno ridefinisce un nome che era già definito in un blocco esterno
- In tal caso, la nuova definizione nasconde (maschera) la vecchia, finché non si esce dal blocco interno

In C:

```
void f(int n) {
    int i = 0;
{
    int n = 0; // maschera n
    n++;        // incrementa la nuova n
}
n--;          // decrementa la vecchia n (il par. formale)
```



Mascheramento

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di . . .

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;  
    DECLARE X, Y;  
BEGIN P  
    ...  
        {X e Y da P}
```



Mascheramento

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

Esempio di scoping
statico

Mascheramento

Mascheramento

Legami di locazione

```
PROGRAM P;  
    DECLARE X, Y;  
BEGIN P  
    ...           {X e Y da P}  
    BLOCK A;  
        DECLARE X, Z;  
    BEGIN A  
        ...           {X e Z da A, Y da P}
```



Mascheramento

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Scope

Necessità dei blocchi

Definizioni

Ambito di ...

Esempio di scoping
statico

Mascheramento

Mascheramento

```
PROGRAM P;
    DECLARE X, Y;
BEGIN P
    ...
    {X e Y da P}
    BLOCK A;
        DECLARE X, Z;
    BEGIN A
        ...
        {X e Z da A, Y da P}
    END A;
    ...
    {X e Y da P}
END P;
```



Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

Legami di locazione



Legami di locazione e scope

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

- Stabilire un legame di locazione si chiama *allocare memoria*
- Se una variabile è in scope, deve avere un legame di locazione
- Non vale il viceversa: un legame di locazione può essere attivo mentre la variabile non è in scope (ad es., per mascheramento)
- Quindi, lo scope (ambito di visibilità) di una variabile è un sottoinsieme dell'ambito di validità del suo legame di locazione



Tecniche di allocazione di memoria

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

- Statica: l'indirizzo viene fissato a load-time e tale allocazione vale per tutta l'esecuzione
- Dinamica
 - ◆ Su stack: l'indirizzo viene fissato a runtime su richiesta (ad es., all'inizio di un blocco) e viene rilasciato al termine del blocco corrente
 - ◆ Su heap: l'indirizzo viene fissato a runtime su richiesta e rilasciato su richiesta oppure automaticamente quando non più utilizzato (garbage collection)



Allocazione statica di memoria

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

Si dice *allocazione statica di memoria* quando il legame di locazione è fissato e costante al tempo di caricamento (load-time)



Allocazione statica di memoria

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

Si dice *allocazione statica di memoria* quando il legame di locazione è fissato e costante al tempo di caricamento (load-time)
Tipico delle variabili globali (se previste dal linguaggio)



Alloc. statica in C

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

In C:

```
int g; // g: alloc. statica e scope globale

void f(int n) {
    static int a; // a: alloc. statica e scope locale
    int i;
    ...
}
```



Allocazione dinamica di memoria su stack

- Si dice *allocazione dinamica di memoria tramite stack* quando il legame di locazione è creato all'inizio dell'esecuzione di un blocco e viene rilasciato automaticamente a fine blocco

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio



Allocazione dinamica di memoria su stack

- Si dice *allocazione dinamica di memoria tramite stack* quando il legame di locazione è creato all'inizio dell'esecuzione di un blocco e viene rilasciato automaticamente a fine blocco
- Essa è realizzata attraverso il *Record di Attivazione (RdA)* di un blocco

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio



Allocazione dinamica di memoria su stack

- Si dice *allocazione dinamica di memoria tramite stack* quando il legame di locazione è creato all'inizio dell'esecuzione di un blocco e viene rilasciato automaticamente a fine blocco
- Essa è realizzata attraverso il *Record di Attivazione (RdA)* di un blocco
- L'RdA contiene tutte le informazioni necessarie all'esecuzione del blocco e alla prosecuzione dell'esecuzione dopo la fine del blocco

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio



L'RdA di un blocco anonimo

- Contenuto dell'RdA nel caso di un blocco anonimo o *in-line*:
 - ◆ Puntatore di catena dinamica (link all'RdA precedente)
 - ◆ Ambiente locale (variabili locali e spazio per risultati intermedi)

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio



L'RdA di un blocco anonimo

- Contenuto dell'RdA nel caso di un blocco anonimo o *in-line*:
 - ◆ Puntatore di catena dinamica (link all'RdA precedente)
 - ◆ Ambiente locale (variabili locali e spazio per risultati intermedi)
- Quando invece il blocco è più complesso (ad es., è una procedura e può essere invocato), l'RdA è più complesso, come si vedrà in seguito

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio



Alloc. su stack in C

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

In C:

```
int g;

void f(int n) { // n: alloc. dinamica su stack e scope locale
    static int a;
    int i;          // i: alloc. dinamica su stack e scope locale
    {
        int j;      // j: alloc. dinamica su stack e scope locale
    }
    ...
}
```



Stack di esecuzione

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

In ogni momento dell'esecuzione lo *stack* (o *pila*) di esecuzione contiene gli RdA “attivi”:



Stack di esecuzione

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

In ogni momento dell'esecuzione lo *stack* (o *pila*) di esecuzione contiene gli RdA “attivi”:

1. il top dello stack contiene l'RdA del blocco correntemente in esecuzione;



Stack di esecuzione

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

In ogni momento dell'esecuzione lo *stack* (o *pila*) di esecuzione contiene gli RdA “attivi”:

1. il top dello stack contiene l'RdA del blocco correntemente in esecuzione;
2. ogni volta che si entra in un blocco, l'RdA del blocco viene creato e posto sullo stack (push);



Stack di esecuzione

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

In ogni momento dell'esecuzione lo *stack* (o *pila*) di esecuzione contiene gli RdA “attivi”:

1. il top dello stack contiene l'RdA del blocco correntemente in esecuzione;
2. ogni volta che si entra in un blocco, l'RdA del blocco viene creato e posto sullo stack (push);
3. ogni volta che si esce da un blocco, viene eliminato l'RdA in cima allo stack (pop)



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

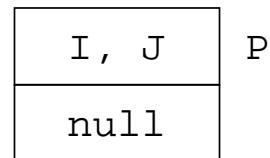
Alloc. su stack in C

Stack di esecuzione

Esempio

All'ingresso in P (supponendo che anche I e J siano allocate sullo stack)

Contenuto dello stack di attivazione



```
PROGRAM P;  
    DECLARE I,J;  
BEGIN P  
    BLOCK A;  
        DECLARE I,K;  
    BEGIN A  
        BLOCK B;  
            DECLARE I,L;  
        BEGIN B  
            ...           {I e L da B, K da A, J da P}  
        END B;  
        ...           {I e K da A, J da P}  
    END A;  
    BLOCK C;  
        DECLARE I,N;  
    BEGIN C  
        ...           {I e N da C, J da P}  
    END C;  
    ...           {I e J da P}  
END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

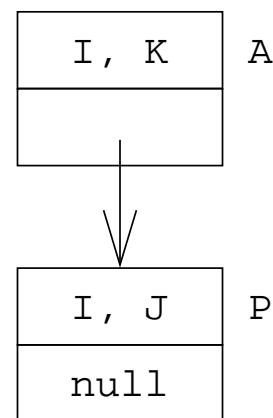
Alloc. su stack in C

Stack di esecuzione

Esempio

All'ingresso in A

Contenuto dello
stack di attivazione



```
PROGRAM P;  
  DECLARE I,J;  
BEGIN P  
  BLOCK A;  
    DECLARE I,K;  
    BEGIN A  
      BLOCK B;  
        DECLARE I,L;  
        BEGIN B  
          ...           { I e L da B, K da A, J da P }  
        END B;  
        ...           { I e K da A, J da P }  
      END A;  
      BLOCK C;  
      DECLARE I,N;  
      BEGIN C  
        ...           { I e N da C, J da P }  
      END C;  
      ...           { I e J da P }  
    END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

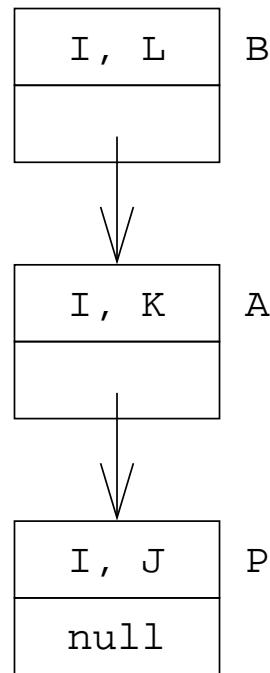
Alloc. su stack in C

Stack di esecuzione

Esempio

All'ingresso in B

Contenuto dello stack di attivazione



```
PROGRAM P;  
  DECLARE I,J;  
BEGIN P  
  BLOCK A;  
    DECLARE I,K;  
  BEGIN A  
    BLOCK B;  
      DECLARE I,L;  
    BEGIN B  
      ... {I e L da B, K da A, J da P}  
    END B;  
    ... {I e K da A, J da P}  
  END A;  
  BLOCK C;  
    DECLARE I,N;  
  BEGIN C  
    ... {I e N da C, J da P}  
  END C;  
  ... {I e J da P}  
END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

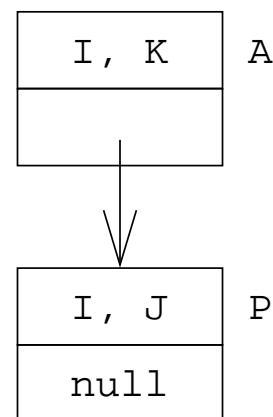
Alloc. su stack in C

Stack di esecuzione

Esempio

All'uscita da B

Contenuto dello
stack di attivazione



```
PROGRAM P;  
  DECLARE I,J;  
BEGIN P  
  BLOCK A;  
    DECLARE I,K;  
    BEGIN A  
      BLOCK B;  
        DECLARE I,L;  
        BEGIN B  
          ... {I e L da B, K da A, J da P}  
        END B;  
        ... {I e K da A, J da P}  
      END A;  
      BLOCK C;  
        DECLARE I,N;  
        BEGIN C  
          ... {I e N da C, J da P}  
        END C;  
        ... {I e J da P}  
      END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

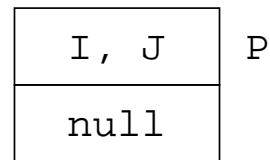
Alloc. su stack in C

Stack di esecuzione

Esempio

All'uscita da A

Contenuto dello
stack di attivazione



```
PROGRAM P;
  DECLARE I,J;
BEGIN P
  BLOCK A;
    DECLARE I,K;
  BEGIN A
    BLOCK B;
      DECLARE I,L;
    BEGIN B
      ...
      {I e L da B, K da A, J da P}
    END B;
    ...
    {I e K da A, J da P}
  END A;
  BLOCK C;
    DECLARE I,N;
  BEGIN C
    ...
    {I e N da C, J da P}
  END C;
  ...
  {I e J da P}
END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

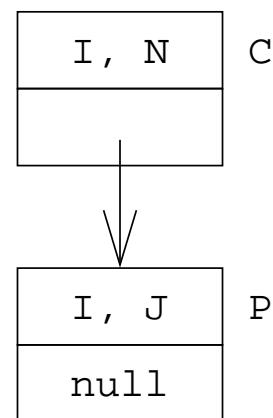
Alloc. su stack in C

Stack di esecuzione

Esempio

All'ingresso in C

Contenuto dello stack di attivazione



```
PROGRAM P;  
  DECLARE I,J;  
BEGIN P  
  BLOCK A;  
    DECLARE I,K;  
  BEGIN A  
    BLOCK B;  
      DECLARE I,L;  
    BEGIN B  
      ... {I e L da B, K da A, J da P}  
    END B;  
    ... {I e K da A, J da P}  
  END A;  
  BLOCK C;  
    DECLARE I,N;  
  BEGIN C  
    ... {I e N da C, J da P}  
  END C;  
  ... {I e J da P}  
END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

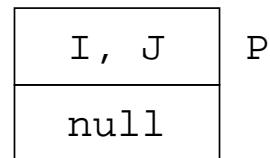
Alloc. su stack in C

Stack di esecuzione

Esempio

All'uscita da C

Contenuto dello
stack di attivazione



```
PROGRAM P;
  DECLARE I,J;
BEGIN P
  BLOCK A;
    DECLARE I,K;
  BEGIN A
    BLOCK B;
      DECLARE I,L;
    BEGIN B
      ...
      {I e L da B, K da A, J da P}
    END B;
    ...
    {I e K da A, J da P}
  END A;
  BLOCK C;
    DECLARE I,N;
  BEGIN C
    ...
    {I e N da C, J da P}
  END C;
  ...
  {I e J da P}
END P;
```



Esempio di allocazione dinamica su stack

Modello imperativo

Data Object e
legami

Legami di tipo

Blocchi di istruzioni

Legami di locazione

Legami di locazione
e scope

Allocazione

Alloc. statica

Alloc. statica in C

Alloc. dinamica

RdA di blocco

Alloc. su stack in C

Stack di esecuzione

Esempio

All'uscita da P

Contenuto dello
stack di attivazione

```
PROGRAM P;
    DECLARE I,J;
BEGIN P
    BLOCK A;
        DECLARE I,K;
BEGIN A
    BLOCK B;
        DECLARE I,L;
BEGIN B
    ...
    {I e L da B, K da A, J da P}
END B;
    ...
    {I e K da A, J da P}
END A;
BLOCK C;
    DECLARE I,N;
BEGIN C
    ...
    {I e N da C, J da P}
END C;
    ...
    {I e J da P}
END P;
```



[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di locazione](#)

Esercizi

[Esercizio 1](#)

[Esercizio 2](#)

[Bibliografia](#)

Esercizi



Esercizio 1

Si considerino i seguenti blocchi annidati. Per ciascun blocco, determinare i legami di ogni variabile. Evidenziare le variabili locali e non locali.

```
PROGRAM P;
  BLOCK B1;
    DECLARE A,B,C;
  BEGIN B1
    BLOCK B2;
      DECLARE C,D;
    BEGIN B2
      BLOCK B3;
        DECLARE B,D,F;
      BEGIN
        ...
      END B3;
      ...
    END B2;
    BLOCK B4;
      DECLARE B,C,D;
    BEGIN B4
      ...
    END B4;
    ...
  END B1;
END P;
```



Esercizio 2

Tracciare il contenuto dello stack di esecuzione durante l'esecuzione del seguente pseudo-programma.

```
PROGRAM P;  
    DECLARE X,Y;  
BEGIN P  
    BLOCK A;  
        DECLARE X,Y,Z;  
    BEGIN A  
        BLOCK B;  
            DECLARE Y;  
        BEGIN B  
            BLOCK C;  
                DECLARE X,Y;  
            BEGIN C  
                ...  
            END C;  
            BLOCK D;  
                DECLARE Z;  
            BEGIN D
```

```
    ...  
    END D;  
    ...  
    END B;  
END A;  
BLOCK E;  
    DECLARE Z;  
BEGIN E  
    BLOCK F;  
        DECLARE X;  
    BEGIN F  
        ...  
    END F;  
    ...  
END E;  
...  
END P;
```



Bibliografia

[Modello imperativo](#)

[Data Object e legami](#)

[Legami di tipo](#)

[Blocchi di istruzioni](#)

[Legami di locazione](#)

[Esercizi](#)

[Esercizio 1](#)

[Esercizio 2](#)

[**Bibliografia**](#)

Capitoli 6 (“I nomi e l’ambiente”) e 7 (“La gestione della memoria”) di *Linguaggi di Programmazione, principi e paradigmi*, di Gabbrielli e Martini (2a edizione)

Linguaggi di Programmazione I

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

30 marzo 2023



Modificatori

Modificatori di accesso

Altri modificatori

Questionario



Modificatori

Modificatori di accesso

Altri modificatori

Questionario

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi



Modificatori

Modificatori di accesso

Altri modificatori

Questionario

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi
- Un gruppo di modificatori, detti *di accesso*, specificano in quali contesti lessicali è visibile quell'elemento



Modificatori

Modificatori di accesso

Altri modificatori

Questionario

- I modificatori sono parole riservate che danno al compilatore informazioni sulla natura del codice, dei dati, delle classi
- Un gruppo di modificatori, detti *di accesso*, specificano in quali contesti lessicali è visibile quell'elemento
- Altri modificatori possono essere usati, in combinazione con i precedenti, per qualificare quell'elemento



Modificatori di accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario

Modificatori di accesso



Generalità

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario

I modificatori di accesso sono:

- public
- protected
- private

Inoltre, se non è presente alcun modificatore, parleremo di “accesso di default”



Generalità

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario

I modificatori di accesso sono:

- public
- protected
- private

Inoltre, se non è presente alcun modificatore, parleremo di “accesso di default”

I seguenti elementi possono essere dotati di modificatore di accesso:

- Classi (anche interfacce ed enumerazioni)
- Attributi
- Metodi e costruttori

In particolare, le variabili locali non supportano modificatori di accesso, perché la loro visibilità è già limitata al blocco corrente



public

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario

- È il modificatore più generoso. Accesso consentito ovunque



public

- È il modificatore più generoso. Accesso consentito ovunque
- Attenzione: l'accesso ad un attributo o un metodo public è subordinato all'accesso alla classe che lo contiene

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (1)

- È il modificatore meno generoso

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (1)

- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (1)

- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

```
public class Complesso {  
    private double reale, immag;  
  
    public Complesso(double r, double i) {  
        reale=r; immag=i;  
    }  
    public Complesso add(Complesso c) {  
        return new Complesso(reale + c.reale,  
                             immag + c.immag);  
    }  
}
```

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (1)

- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

```
public class Complesso {  
    private double reale, immag;  
  
    public Complesso(double r, double i) {  
        reale=r; immag=i;  
    }  
    public Complesso add(Complesso c) {  
        return new Complesso(reale + c.reale,  
                              immag + c.immag);  
    }  
}  
  
public class Cliente {  
    void usali() {  
        Complesso c1 = new Complesso(1, 2);  
        Complesso c2 = new Complesso(3, 4);  
        Complesso c3 = c1.add(c2);  
    }  
}
```

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (1)

- È il modificatore meno generoso
- Può essere usato solo per attributi o metodi, non per classi top-level

```
public class Complesso {  
    private double reale, immag;  
  
    public Complesso(double r, double i) {  
        reale=r; immag=i;  
    }  
    public Complesso add(Complesso c) {  
        return new Complesso(reale + c.reale,  
                               immag + c.immag);  
    }  
}  
  
public class Cliente {  
    void usali() {  
        Complesso c1 = new Complesso(1, 2);  
        Complesso c2 = new Complesso(3, 4);  
        Complesso c3 = c1.add(c2);  
        double d = c3.reale;           // Illegale  
    }  
}
```

Modificatori di accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

```
class Complesso {  
    private double reale, immag;  
}  
  
class SubComplesso extends Complesso {  
    SubComplesso(double r, double i) {  
        reale = r;                                // Illegale  
    }  
}
```

Modificatori di accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

```
class Complesso {  
    private double reale, immag;  
}  
  
class SubComplesso extends Complesso {  
    SubComplesso(double r, double i) {  
        reale = r;                                // Illegale  
    }  
}
```

- La classe SubComplesso eredita gli attributi della superclasse, MA quegli attributi possono essere usati solo dal codice della classe Complesso

Modificatori di accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



private (2)

- Le variabili private possono essere nascoste anche allo stesso oggetto che le possiede

```
class Complesso {  
    private double reale, immag;  
}  
  
class SubComplesso extends Complesso {  
    SubComplesso(double r, double i) {  
        reale = r;                                // Illegale  
    }  
}
```

- La classe SubComplesso eredita gli attributi della superclasse, MA quegli attributi possono essere usati solo dal codice della classe Complesso

Modificatori di accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



Default

- In mancanza di un modificatore di accesso, l'elemento è visibile al codice che si trova nello stesso pacchetto

Modificatori di accesso

Generalità

public

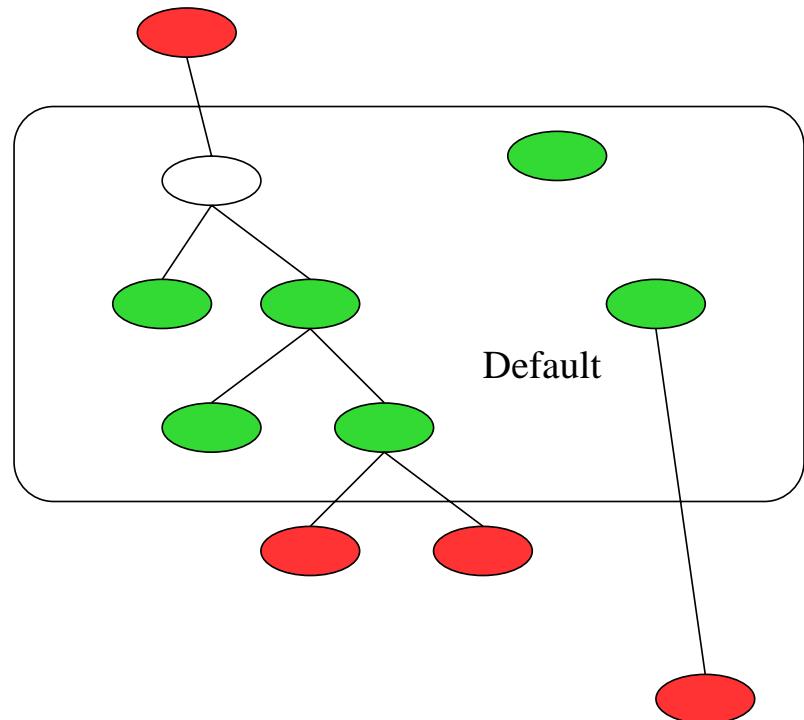
private (1)
private (2)

Default

protected (1)
protected (2)

Altri modificatori

Questionario





protected (1)

- Accesso consentito al codice dello stesso pacchetto e...

Modificatori di accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

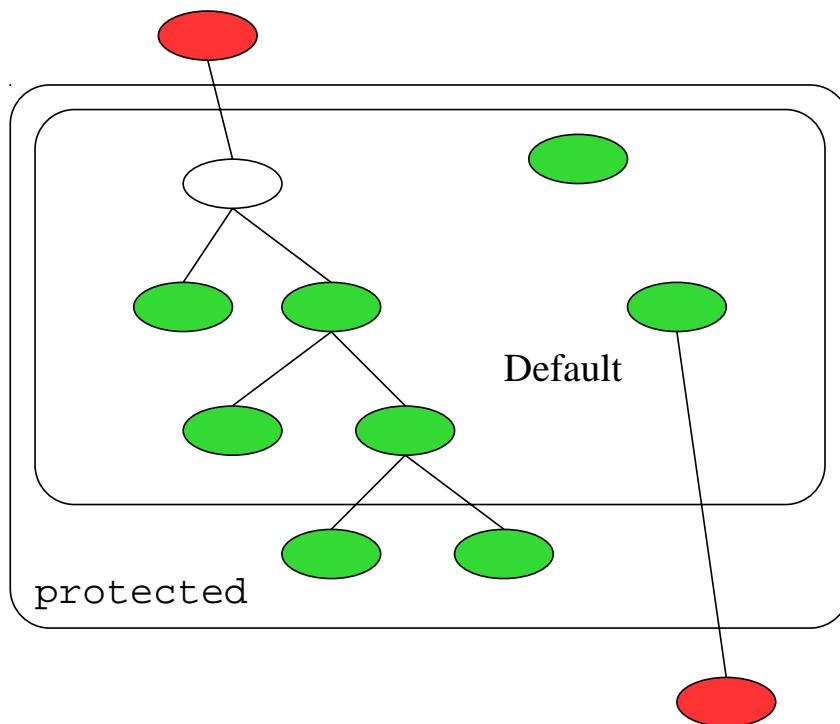
Altri modificatori

Questionario



protected (1)

- Accesso consentito al codice dello stesso pacchetto e...
- ...a tutte le sottoclassi in pacchetti diversi, *ma solo per ereditarietà*
- Una sottoclasse in un altro pacchetto può accedere ad un membro protected della superclasse solo attraverso un riferimento ad un oggetto del proprio tipo (come `this`), o di un suo sottotipo.



Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Altri modificatori

Questionario



protected (2)

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

Esempio:

```
package p1;

public class A {
    protected int x = 7;
}
```

```
package p2;
import p1.A;

public class B extends A {
    public void testProt() {
        System.out.println(x); // Ok: x e' ereditato
        // uso implicito di this
```



protected (2)

Modificatori di
accesso

Generalità

public

private (1)

private (2)

Default

protected (1)

protected (2)

package p1;

```
public class A {  
    protected int x = 7;  
}
```

package p2;

import p1.A;

```
public class B extends A {  
    public void testProt() {  
        System.out.println(x); // Ok: x e' ereditato  
        // uso implicito di this  
        A a = new A();  
        System.out.println(a.x); // Errore di comp.  
        // a non e' di tipo B, ne' di sottotipo di B.  
        B b = new B();  
        System.out.println(b.x); // Ok: accesso  
        // attraverso oggetto di tipo B.  
    }  
}
```



Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario

Altri modificatori



final

- Si applica a classi, metodi e variabili (non a costruttori).

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento final non può essere modificato.

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento final non può essere modificato.
 - ◆ Una classe final non può essere estesa, cioè non può avere sottoclassi.

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento final non può essere modificato.
 - ◆ Una classe final non può essere estesa, cioè non può avere sottoclassi.
 - ◆ Una variabile final è una costante, cioè può solo essere inizializzata.

```
void f(final int n) {  
    final int m = n;  
    n = 0;          // Err. di comp.  
    m = 1;          // Err. di comp.  
    final int a;  
    a = 2;  
}
```

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento final non può essere modificato.
 - ◆ Una classe final non può essere estesa, cioè non può avere sottoclassi.
 - ◆ Una variabile final è una costante, cioè può solo essere inizializzata.

```
void f(final int n) {  
    final int m = n;  
    n = 0;          // Err. di comp.  
    m = 1;          // Err. di comp.  
    final int a;  
    a = 2;  
}
```

- ◆ Un metodo final non può essere sovrascritto (overridden) in una sottoclasse

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



final

- Si applica a classi, metodi e variabili (non a costruttori).
- Il significato varia da contesto a contesto, ma l'essenza è: un elemento final non può essere modificato.
 - ◆ Una classe final non può essere estesa, cioè non può avere sottoclassi.
 - ◆ Una variabile final è una costante, cioè può solo essere inizializzata.

```
void f(final int n) {  
    final int m = n;  
    n = 0;          // Err. di comp.  
    m = 1;          // Err. di comp.  
    final int a;  
    a = 2;  
}
```

- ◆ Un metodo final non può essere sovrascritto (overridden) in una sottoclasse
- ◆ Non ha senso attribuire final a un costruttore, perché esso non è ereditato dalle sottoclassi e quindi non è mai sovrascrivibile

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



final

- Attenzione: un riferimento **final** ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì!

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



final

- Attenzione: un riferimento **final** ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì!

```
final Impiegato luca = new Impiegato("Luca", 1500);
luca = new Impiegato("Luca", 1600); // Err. di comp.
luca.setSalario(1600);           // OK
```

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



final

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di

inizializzazione

static

- Blocchi di

inizializzazione

static

Questionario

- Attenzione: un riferimento **final** ad un oggetto non può essere modificato (cioè riassegnato ad un altro oggetto), ma l'oggetto a cui esso fa riferimento sì!

```
final Impiegato luca = new Impiegato("Luca", 1500);
luca = new Impiegato("Luca", 1600); // Err. di comp.
luca.setSalario(1600);           // OK
```

- Attenzione: un array **final** non può essere riassegnato, ma il contenuto dell'array può essere modificato!

```
final int[] numeri = new int[10];
numeri = new int[20]; // Err. di comp.
numeri[0] = 77;     // OK
```



abstract

- Si applica solo a classi e a metodi.
- Un metodo **abstract** non possiede corpo (";" invece di "{...}"):

```
abstract void getValore();
```

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



abstract

- Si applica solo a classi e a metodi.
- Un metodo **abstract** non possiede corpo (";" invece di "{...}"):

```
abstract void getValore();
```

- Una classe DEVE essere marcata **abstract** se:
 - ◆ essa contiene almeno un metodo **abstract**, oppure

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static
- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



abstract

- Si applica solo a classi e a metodi.
- Un metodo **abstract** non possiede corpo (";" invece di "{...}"):

```
abstract void getValore();
```
- Una classe DEVE essere marcata **abstract** se:
 - ◆ essa contiene almeno un metodo **abstract**, oppure
 - ◆ essa eredita almeno un metodo **abstract** per il quale non fornisce una realizzazione, oppure

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static
- Metodi static

Esempio di binding

- Blocchi di
inizializzazione
- static
- Blocchi di
inizializzazione
- static

Questionario



abstract

- Si applica solo a classi e a metodi.
- Un metodo **abstract** non possiede corpo (";" invece di "{...}"):

```
abstract void getValore();
```
- Una classe DEVE essere marcata **abstract** se:
 - ◆ essa contiene almeno un metodo **abstract**, oppure
 - ◆ essa eredita almeno un metodo **abstract** per il quale non fornisce una realizzazione, oppure
 - ◆ essa dichiara di implementare una interfaccia [vedremo in seguito], ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static
- Metodi static

Esempio di binding

- Blocchi di inizializzazione
- static
- Blocchi di inizializzazione
- static

Questionario



abstract

- Si applica solo a classi e a metodi.
- Un metodo **abstract** non possiede corpo (";" invece di "{...}"):

```
abstract void getValore();
```

- Una classe DEVE essere marcata **abstract** se:
 - ◆ essa contiene almeno un metodo **abstract**, oppure
 - ◆ essa eredita almeno un metodo **abstract** per il quale non fornisce una realizzazione, oppure
 - ◆ essa dichiara di implementare una interfaccia [vedremo in seguito], ma non fornisce una realizzazione di tutti i metodi di quell'interfaccia.
- Qualsiasi classe PUÒ essere marcata **abstract**, anche se non contiene metodi astratti. Il compilatore impedirà di istanziarla.
- In un certo senso, **abstract** è opposto a **final**: una classe (o metodo) **final** non può essere specializzata; una classe (o metodo) **abstract** esiste solo per essere specializzata.

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static
- Metodi static

Esempio di binding

- Blocchi di
inizializzazione
static
- Blocchi di
inizializzazione
static

Questionario



static

- Si applica ad attributi, metodi ed anche a blocchi di codice esterni ai metodi

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



static

- Si applica ad attributi, metodi ed anche a blocchi di codice esterni ai metodi
- In generale, una caratteristica static appartiene alla classe, non alle singole istanze: essa è unica, indipendentemente dal numero (anche zero) di istanze di quella classe.

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



- Attributi static

- L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {  
    static int x = 0;  
    Ecstatic () { x++; }  
}
```

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di

inizializzazione

static

- Blocchi di

inizializzazione

static

Questionario

- Attributi static

- L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {  
    static int x = 0;  
    Ecstatic () { x++; }  
}
```

- L'accesso ad un attributo static di una classe può avvenire (con la dot-notation):
 - ◆ o partendo da un riferimento ad una istanza di quella classe,



Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario

- Attributi static

- L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {  
    static int x = 0;  
    Ecstatic () { x++; }  
}
```

- L'accesso ad un attributo static di una classe può avvenire (con la dot-notation):
 - ◆ o partendo da un riferimento ad una istanza di quella classe,
 - ◆ o partendo dal nome stesso della classe.



Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di

inizializzazione

static

- Blocchi di

inizializzazione

static

Questionario

- Attributi static

- L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {  
    static int x = 0;  
    Ecstatic () { x++; }  
}
```

- L'accesso ad un attributo static di una classe può avvenire (con la dot-notation):
 - ◆ o partendo da un riferimento ad una istanza di quella classe,
 - ◆ o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);
```



- Attributi static

- L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {  
    static int x = 0;  
    Ecstatic () { x++; }  
}
```

- L'accesso ad un attributo static di una classe può avvenire (con la dot-notation):
 - ◆ o partendo da un riferimento ad una istanza di quella classe,
 - ◆ o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);  
Ecstatic e = new Ecstatic();  
e.x = 100;
```

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



- Attributi static

- L'inizializzazione di un attributo static avviene nel momento in cui la classe viene caricata in memoria (anche se non esisterà mai nessuna istanza di quella classe).

```
class Ecstatic {  
    static int x = 0;  
    Ecstatic () { x++; }  
}
```

- L'accesso ad un attributo static di una classe può avvenire (con la dot-notation):
 - ◆ o partendo da un riferimento ad una istanza di quella classe,
 - ◆ o partendo dal nome stesso della classe.

```
System.out.println(Ecstatic.x);  
Ecstatic e = new Ecstatic();  
e.x = 100;  
Ecstatic.x = 100;
```

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di

inizializzazione

static

- Blocchi di

inizializzazione

static

Questionario



- Metodi static

- Appartengono alla classe e non alle singole istanze

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



- Metodi static

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {  
    public static void f() { ... }  
}  
...  
Test.f();
```

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



- Metodi static

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {  
    public static void f() { ... }  
}  
...  
Test.f();
```

- Non posseggono il riferimento this

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



- Metodi static

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {  
    public static void f() { ... }  
}  
...  
Test.f();
```

- Non posseggono il riferimento `this`
- Esempio: il metodo `main`

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



- Metodi static

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {  
    public static void f() { ... }  
}  
...  
Test.f();
```

- Non posseggono il riferimento `this`
- Esempio: il metodo `main`
- Hanno binding statico e non possono essere sovrascritti (`overridden`)

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione

static

- Blocchi di
inizializzazione

static

Questionario



- Metodi static

- Appartengono alla classe e non alle singole istanze
- Si possono invocare a partire dal nome della classe:

```
class Test {  
    public static void f() { ... }  
}  
...  
Test.f();
```

- Non posseggono il riferimento `this`
- Esempio: il metodo `main`
- Hanno binding statico e non possono essere sovrascritti (`overridden`)

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione

static

- Blocchi di inizializzazione

static

Questionario



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}
```

```
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
    public static void main(...){  
        A a = new A();  
        B b = new B();  
        A x = b;  
  
        /* binding statico */  
        A.f(); /* stampa "f di A" */  
        B.f(); /* stampa "f di B" */  
        a.f();
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
public static void main(...){  
    A a = new A();  
    B b = new B();  
    A x = b;  
  
    /* binding statico */  
    A.f(); /* stampa "f di A" */  
    B.f(); /* stampa "f di B" */  
    a.f(); /* stampa "f di A" */  
    b.f();
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
    public static void main(...){  
        A a = new A();  
        B b = new B();  
        A x = b;  
  
        /* binding statico */  
        A.f(); /* stampa "f di A" */  
        B.f(); /* stampa "f di B" */  
        a.f(); /* stampa "f di A" */  
        b.f(); /* stampa "f di B" */  
        x.f();
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
public static void main(...){  
    A a = new A();  
    B b = new B();  
    A x = b;  
  
    /* binding statico */  
    A.f(); /* stampa "f di A" */  
    B.f(); /* stampa "f di B" */  
    a.f(); /* stampa "f di A" */  
    b.f(); /* stampa "f di B" */  
    x.f(); /* stampa "f di A" */  
  
    /* binding dinamico */  
    a.g();
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
public static void main(...){  
    A a = new A();  
    B b = new B();  
    A x = b;  
  
    /* binding statico */  
    A.f(); /* stampa "f di A" */  
    B.f(); /* stampa "f di B" */  
    a.f(); /* stampa "f di A" */  
    b.f(); /* stampa "f di B" */  
    x.f(); /* stampa "f di A" */  
  
    /* binding dinamico */  
    a.g(); /* stampa "g di A" */  
    b.g();
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
public static void main(...){  
    A a = new A();  
    B b = new B();  
    A x = b;  
  
    /* binding statico */  
    A.f(); /* stampa "f di A" */  
    B.f(); /* stampa "f di B" */  
    a.f(); /* stampa "f di A" */  
    b.f(); /* stampa "f di B" */  
    x.f(); /* stampa "f di A" */  
  
    /* binding dinamico */  
    a.g(); /* stampa "g di A" */  
    b.g(); /* stampa "g di B" */  
    x.g();
```



Esempio di binding

```
class A {  
  
    public static void f() {  
        System.out.println("f di A");  
    }  
  
    public void g() {  
        System.out.println("g di A");  
    }  
}  
  
class B extends A {  
  
    // mascheramento, non overriding  
    public static void f() {  
        System.out.println("f di B");  
    }  
  
    @Override  
    public void g() {  
        System.out.println("g di B");  
    }  
}
```

```
class C {  
    public static void main(...){  
        A a = new A();  
        B b = new B();  
        A x = b;  
  
        /* binding statico */  
        A.f(); /* stampa "f di A" */  
        B.f(); /* stampa "f di B" */  
        a.f(); /* stampa "f di A" */  
        b.f(); /* stampa "f di B" */  
        x.f(); /* stampa "f di A" */  
  
        /* binding dinamico */  
        a.g(); /* stampa "g di A" */  
        b.g(); /* stampa "g di B" */  
        x.g(); /* stampa "g di B" */  
    }  
}
```



- Blocchi di inizializzazione static

- Una classe può contenere blocchi di codice marcati static

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di inizializzazione static

- Blocchi di inizializzazione static

Questionario



Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione
static

- Blocchi di
inizializzazione
static

Questionario

- Blocchi di inizializzazione static

- Una classe può contenere blocchi di codice marcati **static**
- Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria



- Blocchi di inizializzazione static

- Una classe può contenere blocchi di codice marcati **static**
- Tali blocchi sono eseguiti una sola volta, nell'ordine in cui compaiono, quando la classe viene caricata in memoria

```
public class EsempioStatic {  
    private static double d=1.23;  
  
    static {  
        System.out.println("Primo blocco static: d=" + d++);  
    }  
  
    public static void main(String[] args) {  
        System.out.println("main: d=" + d++);  
    }  
  
    static {  
        System.out.println("Secondo blocco static: d=" + d++);  
    }  
}
```

Modificatori di
accesso

Altri modificatori

final

final

abstract

static

- Attributi static

- Metodi static

Esempio di binding

- Blocchi di
inizializzazione
static

- Blocchi di
inizializzazione
static

Questionario



- Blocchi di inizializzazione static

- L'uso tipico consiste nell'inizializzare gli attributi statici

```
public class Esempio {  
    private static final int SIZE = 100;  
    private static int[] numeri = new int[SIZE];  
  
    static {  
        for (int i=0; i<SIZE; i++) {  
            numeri[i] = i;  
        }  
    }  
    ...  
}
```

Modificatori di accesso

Altri modificatori

final

final

abstract

static

- Attributi statici

- Metodi statici

Esempio di binding

- Blocchi di

inizializzazione

static

- Blocchi di
inizializzazione
static

Questionario



Modificatori di
accesso

Altri modificatori

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

Questionario



D 1

Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

A.

```
Integer x = new Integer(100);
Integer y = new Integer(100);
if (x == y) {
    System.out.println("Uguale");
}
```

B.

```
int x=100;
Integer y = new Integer(100);
if (x == y) {
    System.out.println("Uguale");
}
```

C.

```
int x=100; float y=100.0F;
if (x == y) {
    System.out.println("Uguale");
}
```

D.

```
String x = new String("100");
String y = new String("100");
if (x == y) {
    System.out.println("Uguale");
}
```

E.

```
String x = "100";
String y = "100";
if (x == y) {
    System.out.println("Uguale");
}
```



D 1

Quale dei seguenti frammenti viene correttamente compilato e stampa "Uguale" in esecuzione?

A.

```
Integer x = new Integer(100);
Integer y = new Integer(100);
if (x == y) {
    System.out.println("Uguale");
}
```

B.

```
int x=100;
Integer y = new Integer(100);
if (x == y) {
    System.out.println("Uguale");
}
```

C.

```
int x=100; float y=100.0F;
if (x == y) {
    System.out.println("Uguale");
}
```

D.

```
String x = new String("100");
String y = new String("100");
if (x == y) {
    System.out.println("Uguale");
}
```

E.

```
String x = "100";
String y = "100";
if (x == y) {
    System.out.println("Uguale");
}
```

C. E. – (E) funziona a causa di una ottimizzazione del compilatore, che riusa lo stesso oggetto quando vede due litterali String uguali.



D 2

Quali delle seguenti dichiarazioni sono illegali?

- A. default String s;
- B. transient int i = 41;
- C. public final static native int w();
- D. abstract double d;
- E. abstract final double cosenoIpertbolico();

[Modificatori di accesso](#)

[Altri modificatori](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)



D 2

Quali delle seguenti dichiarazioni sono illegali?

- A. default String s;
- B. transient int i = 41;
- C. public final static native int w();
- D. abstract double d;
- E. abstract final double cosenoIpertbolico();

A. D. E.

[Modificatori di accesso](#)

[Altri modificatori](#)

[Questionario](#)

[D 1](#)

[D 2](#)

[D 3](#)

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)



D 3

Quale delle seguenti affermazioni è vera?

- A. Una classe **abstract** non può avere metodi **final**.
- B. Una classe **final** non può avere metodi **abstract**.

[Modificatori di accesso](#)

[Altri modificatori](#)

[Questionario](#)

[D 1](#)

[D 2](#)

D 3

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)



D 3

Quale delle seguenti affermazioni è vera?

- A. Una classe **abstract** non può avere metodi **final**.
 - B. Una classe **final** non può avere metodi **abstract**.
-
- B. – Una classe che contiene metodi **abstract** deve essere anch'essa **abstract**, ma ciò è in contraddizione con il modificatore **final**.

[Modificatori di accesso](#)

[Altri modificatori](#)

[Questionario](#)

[D 1](#)

[D 2](#)

D 3

[D 4](#)

[D 5](#)

[D 6](#)

[D 7](#)

[D 8](#)

[D 9](#)

[D 10](#)



D 4

Qual è la minima modifica che rende il seguente codice compilabile?

```
1. final class Aaa {  
2.     int xxx;  
3.     void yyy() { xxx=1; }  
4. }  
5.  
6. class Bbb extends Aaa {  
7.     final Aaa fref = new Aaa();  
8.     final void yyy() {  
9.         System.out.println(  
10.            "In yyy()");  
11.         fref.xxx = 12345;  
12.     }  
13. }
```

- A. Alla linea 1, rimuovere final.
 - B. Alla linea 7, rimuovere final.
 - C. Rimuovere la linea 11.
 - D. Alle linee 1 e 7, rimuovere final.
 - E. Nessuna modifica è necessaria.
-



D 4

Qual è la minima modifica che rende il seguente codice compilabile?

```
1. final class Aaa {  
2.     int xxx;  
3.     void yyy() { xxx=1; }  
4. }  
5.  
6. class Bbb extends Aaa {  
7.     final Aaa fref = new Aaa();  
8.     final void yyy() {  
9.         System.out.println(  
10.            "In yyy()");  
11.         fref.xxx = 12345;  
12.     }  
13. }
```

- A. Alla linea 1, rimuovere final.
- B. Alla linea 7, rimuovere final.
- C. Rimuovere la linea 11.
- D. Alle linee 1 e 7, rimuovere final.
- E. Nessuna modifica è necessaria.

A.



D 5

Riguardo al codice seguente, quale affermazione è vera?

```
1. class Roba {  
2.     static int x = 10;  
3.     static { x += 5; }  
4.  
5.     public static void main(String[] args) {  
6.         System.out.println("x=" + x);  
7.     }  
8.  
9.     static { x /= 5; }  
10. }
```

- A. Le linee 3 e 9 non sono compilate, poiché mancano i nomi di metodi e i tipi di ritorno.
 - B. La linea 9 non è compilata, poiché si può avere solo un blocco top-level static.
 - C. Il codice viene compilato e l'esecuzione produce x=10.
 - D. Il codice viene compilato e l'esecuzione produce x=15.
 - E. Il codice viene compilato e l'esecuzione produce x=3.
-



D 5

Riguardo al codice seguente, quale affermazione è vera?

```
1. class Roba {  
2.     static int x = 10;  
3.     static { x += 5; }  
4.  
5.     public static void main(String[] args) {  
6.         System.out.println("x=" + x);  
7.     }  
8.  
9.     static { x /= 5; }  
10. }
```

- A. Le linee 3 e 9 non sono compilate, poiché mancano i nomi di metodi e i tipi di ritorno.
- B. La linea 9 non è compilata, poiché si può avere solo un blocco top-level static.
- C. Il codice viene compilato e l'esecuzione produce x=10.
- D. Il codice viene compilato e l'esecuzione produce x=15.
- E. Il codice viene compilato e l'esecuzione produce x=3.

E.



D 6

Rispetto al codice seguente, quale affermazione è vera?

```
1. class A {  
2.     private static int x=100;  
3.  
4.     public static void main(  
5.         String[] args) {  
6.         A hs1 = new A();  
7.         hs1.x++;  
8.         A hs2 = new A();  
9.         hs2.x++;  
10.        hs1 = new A();  
11.        hs1.x++;  
12.        A.x++;  
13.        System.out.println(  
14.            "x = " + x);  
15.    }  
16. }
```

- A. La linea 7 non compila, poiché è un riferimento static ad una variabile private.
- B. La linea 12 non compila, poiché è un riferimento static ad una variabile private.
- C. Il programma viene compilato e stampa x = 102.
- D. Il programma viene compilato e stampa x = 103.
- E. Il programma viene compilato e stampa x = 104.



D 6

Rispetto al codice seguente, quale affermazione è vera?

```
1. class A {  
2.     private static int x=100;  
3.  
4.     public static void main(  
5.         String[] args) {  
6.         A hs1 = new A();  
7.         hs1.x++;  
8.         A hs2 = new A();  
9.         hs2.x++;  
10.        hs1 = new A();  
11.        hs1.x++;  
12.        A.x++;  
13.        System.out.println(  
14.            "x = " + x);  
15.    }  
16. }
```

- A. La linea 7 non compila, poiché è un riferimento static ad una variabile private.
- B. La linea 12 non compila, poiché è un riferimento static ad una variabile private.
- C. Il programma viene compilato e stampa x = 102.
- D. Il programma viene compilato e stampa x = 103.
- E. Il programma viene compilato e stampa x = 104.

E.



D 7

Modificatori di
accesso

Altri modificatori

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

Dato il codice seguente:

```
1. class SuperC {  
2.     void unMetodo() { }  
3. }  
4.  
5. class SubC extends SuperC {  
6.     void unMetodo() { }  
7. }
```

1. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 2, lasciando il resto del codice inalterato?
2. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 6, lasciando il resto del codice inalterato?



D 7

Modificatori di
accesso

Altri modificatori

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

Dato il codice seguente:

```
1. class SuperC {  
2.     void unMetodo() { }  
3. }  
4.  
5. class SubC extends SuperC {  
6.     void unMetodo() { }  
7. }
```

1. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 2, lasciando il resto del codice inalterato?
 2. Quali modificatori di accesso possono essere legalmente dati ad `unMetodo` alla linea 6, lasciando il resto del codice inalterato?
-
1. Alla linea 2, il metodo può essere `private`, visto che nella sottoclasse viene sovrapposto da un metodo con visibilità di pacchetto.
 2. Alla linea 6, il metodo può essere `protected` oppure `public`, visto che sovrappone un metodo con visibilità di pacchetto.



D 8

```
1. package abcd;  
2.  
3. public class SupA {  
4.     protected static int count=0;  
5.     public SupA() { count++; }  
6.     protected void f() {}  
7.     static int getCount() {  
8.         return count;  
9.     }  
10. }
```

```
1. package abcd;  
2.  
3. class A extends abcd.SupA {  
4.     public void f() {}  
5.     public int getCount() {  
6.         return count;  
7.     }  
8. }
```

Riguardo ai codici a sinistra, quale affermazione è vera?

- A. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e A è nello stesso pacchetto di SupA.
- B. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e public nella sottoclasse.
- C. La compilazione di A.java fallisce alla linea 5, poiché il metodo getCount() è static nella superclasse e non può essere sovrapposto da un metodo non-static.
- D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo f() su una istanza di A.
- E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo getCount su una istanza di SupA.



D 8

```
1. package abcd;  
2.  
3. public class SupA {  
4.     protected static int count=0;  
5.     public SupA() { count++; }  
6.     protected void f() {}  
7.     static int getCount() {  
8.         return count;  
9.     }  
10. }
```

```
1. package abcd;  
2.  
3. class A extends abcd.SupA {  
4.     public void f() {}  
5.     public int getCount() {  
6.         return count;  
7.     }  
8. }
```

Riguardo ai codici a sinistra, quale affermazione è vera?

- A. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e A è nello stesso pacchetto di SupA.
- B. La compilazione di A.java fallisce alla linea 4, poiché il metodo f() è protected nella superclasse e public nella sottoclasse.
- C. La compilazione di A.java fallisce alla linea 5, poiché il metodo getCount() è static nella superclasse e non può essere sovrapposto da un metodo non-static.
- D. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo f() su una istanza di A.
- E. I codici sono compilati, ma viene lanciata una eccezione quando viene invocato il metodo getCount su una istanza di SupA.

C.



D 9

Riguardo ai codici seguenti, quale affermazione è vera?

```
1. package abcd;  
2.  
3. public class SupA {  
4.     protected static int count=0;  
5.     public SupA() { count++; }  
6.     protected void f() {}  
7.     static int getCount() {  
8.         return count;  
9.     }  
10. }
```

```
8.     System.out.print(  
9.             "Prima:" + count);  
10.    A a = new A();  
11.    System.out.println(  
12.            " Dopo:" + count);  
13.    a.f();  
14. }
```

- A. Il programma viene compilato e stampa:
Prima:0 Dopo:2
- B. Il programma viene compilato e stampa:
Prima:0 Dopo:1
- C. La compilazione di A fallisce alla linea 4.
- D. La compilazione di A fallisce alla linea 13.
- E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 13.

```
1. package ab;  
2.  
3. class A extends abcd.SupA {  
4.     A() { count++; }  
5.  
6.     public static void main(  
7.         String[] args) {
```



D 9

Riguardo ai codici seguenti, quale affermazione è vera?

```
1. package abcd;  
2.  
3. public class SupA {  
4.     protected static int count=0;  
5.     public SupA() { count++; }  
6.     protected void f() {}  
7.     static int getCount() {  
8.         return count;  
9.     }  
10. }
```

```
8.     System.out.print(  
9.             "Prima:" + count);  
10.    A a = new A();  
11.    System.out.println(  
12.            " Dopo:" + count);  
13.    a.f();  
14. }
```

```
1. package ab;  
2.  
3. class A extends abcd.SupA {  
4.     A() { count++; }  
5.  
6.     public static void main(  
7.         String[] args) {
```

- A. Il programma viene compilato e stampa:
Prima:0 Dopo:2
- B. Il programma viene compilato e stampa:
Prima:0 Dopo:1
- C. La compilazione di A fallisce alla linea 4.
- D. La compilazione di A fallisce alla linea 13.
- E. Il programma viene compilato, ma viene lanciata una eccezione alla linea 13.

A.



D 10

Modificatori di
accesso

Altri modificatori

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

Si consideri la classe seguente:

```
1. public class Test1 {  
2.     public float unMetodo(float a, float b) {  
3.         }  
4.     }  
5. }
```

Quali dei seguenti metodi possono lecitamente essere inseriti alla linea 4?

- A. public int unMetodo(int a, int b) { }
- B. public float unMetodo(float a, float b) { }
- C. public float unMetodo(float a, float b, int c) { }
- D. public float unMetodo(float c, float d) { }
- E. private float unMetodo(int a, int b, int c) { }



D 10

Modificatori di
accesso

Altri modificatori

Questionario

D 1

D 2

D 3

D 4

D 5

D 6

D 7

D 8

D 9

D 10

Si consideri la classe seguente:

```
1. public class Test1 {  
2.     public float unMetodo(float a, float b) {  
3.         }  
4.     }  
5. }
```

Quali dei seguenti metodi possono lecitamente essere inseriti alla linea 4?

- A. public int unMetodo(int a, int b) { }
- B. public float unMetodo(float a, float b) { }
- C. public float unMetodo(float a, float b, int c) { }
- D. public float unMetodo(float c, float d) { }
- E. private float unMetodo(int a, int b, int c) { }

A. C. E. – B e D potrebbero essere overriding non overloading.

Linguaggi di Programmazione I – Lezione 17

Proff. Piero Bonatti e Marco Faella

<mailto://pab@unina.it>

<mailto://marfaella@gmail.com>

25 maggio 2022



Panoramica dell'argomento

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Paradigma funzionale



L'essenza del paradigma funzionale

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive

Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



L'essenza del paradigma funzionale

Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
 - ◆ perché gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare)



L'essenza del paradigma funzionale

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
 - ◆ perché gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare)
- Quindi, senza assegnamenti, non ci possono essere cicli while/for

Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



L'essenza del paradigma funzionale

Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
 - ◆ perché gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare)
- Quindi, senza assegnamenti, non ci possono essere cicli while/for
- Conseguenze sulla programmazione:
 - ◆ ricorsione al posto dei cicli
 - ◆ modifiche all'ambiente anzichè alla memoria:
 - ◆ creazione identificatori con stesso nome mediante chiamate ricorsive
 - ◆ creazione di nuovi identificatori con lo stesso nome che mascherano la versione precedente (come nello scoping statico)



Introduzione

- In questo corso illustreremo brevemente i linguaggi funzionali:
 - ◆ ML (Meta Language, nella versione *Standard ML of New Jersey*, www.smlnj.org)
 - ◆ Altri linguaggi funzionali: OCaml, F#; Lisp, Scheme, Haskell, Scala
- I linguaggi OO più diffusi stanno acquistando caratteristiche funzionali (prossime slide)
- Testi di riferimento
 - ◆ Capitolo 13 di: Gabbielli e Martini, *Linguaggi di Programmazione* (2a ed.)
 - ◆ Riccardo Pucella, *Notes on Programming Standard ML of New Jersey*.
<https://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf>



Costrutti funzionali in Java

Paradigma
funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

- Lambda espressioni (funzioni anonime) [Java 8]
- Collezioni funzionali (*stream*) [Java 8]
- Type inference per metodi parametrici [Java 5]
- Type inference per variabili locali (keyword var) [Java 10]
- Pattern matching su instanceof [Java 16]
- Pattern matching su switch
[Java 17, preview]
- Classi immutabili (record) [Java 16]



Costrutti funzionali in C++

Paradigma
funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

■ Lambda espressioni (funzioni anonime)

[C++11]

■ Type inference (keyword auto)

[C++11]



Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

ML



Il sistema di tipi

- ML è *fortemente e staticamente tipato*
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il sistema di tipi

- ML è *fortemente* e *staticamente* tipato
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
 - ◆ spesso lo capisce da solo (*type inference*)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il sistema di tipi

- ML è *fortemente e staticamente tipato*
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
 - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il sistema di tipi

- ML è *fortemente e staticamente tipato*
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
 - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il sistema di tipi

- ML è *fortemente e staticamente tipato*
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
 - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il sistema di tipi

- ML è *fortemente e staticamente tipato*
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
 - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)
- Supporta *encapsulation* (tipi di dato astratti) ma non è un linguaggio a oggetti
 - ◆ mancano la gerarchia di tipi e – di conseguenza – l'ereditarietà

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il sistema di tipi

- ML è *fortemente e staticamente tipato*
 - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
 - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)
- Supporta *encapsulation* (tipi di dato astratti) ma non è un linguaggio a oggetti
 - ◆ mancano la gerarchia di tipi e – di conseguenza – l'ereditarietà
- Il linguaggio OCaml supporta anche gerarchie di tipi ed ereditarietà (è object-oriented)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Implementazioni di ML

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

ML può essere usato in 2 modi:

1. Interagendo con l'interprete (ad es. **Standard ML of New Jersey**)

- inserendo definizioni ed espressioni una per una
- l'interprete risponde ad ogni passo
- si possono caricare programmi da file digitando nella shell dell'interprete il comando

```
use "nome del file";
```

questo rende utilizzabili le dichiarazioni contenute nel file



Implementazioni di ML

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

ML può essere usato in 2 modi:

1. Interagendo con l'interprete (ad es. **Standard ML of New Jersey**)

- inserendo definizioni ed espressioni una per una
- l'interprete risponde ad ogni passo
- si possono caricare programmi da file digitando nella shell dell'interprete il comando

```
use "nome del file";
```

questo rende utilizzabili le dichiarazioni contenute nel file

2. Compilando un programma in codice oggetto direttamente eseguibile

- ad es. mediante il compilatore **mlton** per standard ML

```
mlton "nome del file.sml"
```

questo comando produce un file eseguibile con lo stesso nome (ma senza l'estensione .sml)



I tipi primitivi (I)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Detti anche *base types*

■ **int**: gli interi

0, 1, ~1, 2, ~2, ... 0xff, ~0x32, ...

Notare che si usa ~ invece del segno meno (-). Alcuni operatori:

+, -, *, div, mod, =, <, ...



I tipi primitivi (I)

Detti anche *base types*

■ **int**: gli interi

```
0, 1, ~1, 2, ~2, ...          0xff, ~0x32, ...
```

Notare che si usa `~` invece del segno meno (`-`). Alcuni operatori:

```
+, -, *, div, mod, =, <, ...
```

■ **word**: unsigned integers, prefisso 0w

```
0w44, 0w15, ...          0wxff, ...
```

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



I tipi primitivi (I)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Detti anche *base types*

■ **int**: gli interi

0, 1, ~1, 2, ~2, ... 0xff, ~0x32, ...

Notare che si usa ~ invece del segno meno (-). Alcuni operatori:

+, -, *, div, mod, =, <, ...

■ **word**: unsigned integers, prefisso 0w

0w44, 0w15, ... 0wxff, ...

■ **real**

3.14, 2.0, 0.1E6, ...

Alcuni operatori su real:

+, -, *, /, <, ...



I tipi primitivi (II)

■ string

```
"abc", "123", ...
```

Alcuni operatori su string:

```
^, size, =, <, ...
```

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



I tipi primitivi (II)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ string

```
"abc", "123", ...
```

Alcuni operatori su string:

```
^, size, =, <, ...
```

■ char

```
#"a", #"\n", #"\163", ...
```

Alcuni operatori su char:

```
ord, chr, =, <, ...
```



I tipi primitivi (II)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ string

```
"abc", "123", ...
```

Alcuni operatori su string:

```
^, size, =, <, ...
```

■ char

```
#"a", #"\n", #"\163", ...
```

Alcuni operatori su char:

```
ord, chr, =, <, ...
```

■ bool

```
true, false
```

Alcuni operatori su bool:

```
not, andalso, orelse, =
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
-
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml  
Standard ML of New Jersey v110.79  
- 3;
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4;
val it = 0wx3 : word
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4;
val it = 0wx3 : word

- "Hello" ^ "world";
val it = "Hello world" : string
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4;
val it = 0wx3 : word

- "Hello" ^ "world";
val it = "Hello world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4;
val it = 0wx3 : word

- "Hello" ^ "world";
val it = "Hello world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int

- 3 + 2.2;
Error: operator and operand don't agree [overload conflict]
```



Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- 0w7 mod 0w4;
val it = 0wx3 : word

- "Hello" ^ "world";
val it = "Hello world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int

- 3 + 2.2;
Error: operator and operand don't agree [overload conflict]

- real(3) + 2.2;
val it = 5.2 : real
```

■ 'it' si riferisce all'espressione data; calcola sia valore che tipo



I tipi primitivi (III)

- *Nessuna conversione automatica tra tipi numerici! Usare real:int->real e basis library*

```
- val r = 3.0 + 2;  
Error: operator and operand don't agree  
  
- val r = 3.0 + real(2);  
val r = 5.0 : real  
  
- val i = 1 + 0w1;  
Error: operator and operand don't agree  
  
- val i = 1 + Word.toInt(0w1);  
val i = 2 : int
```

- C'è una basis library per ogni tipo primitivo (Int, Word, Real...) con funzioni per conversioni, parsing, e altre utilità

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



I tipi primitivi (IV)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Real non supporta l'uguaglianza! Usare Real.==

```
- val x = 1.0; val y = 2.0;  
val x = 1.0 : real  
val y = 2.0 : real  
- x = y;
```

Error: operator and operand don't agree [equality type required]

```
- Real.==(x,y);  
val it = false : bool
```



I tipi primitivi (IV)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Real non supporta l'uguaglianza! Usare Real.==

```
- val x = 1.0; val y = 2.0;  
val x = 1.0 : real  
val y = 2.0 : real  
- x = y;  
Error: operator and operand don't agree [equality type required]  
  
- Real.==(x,y);  
val it = false : bool
```

- Questo perchè lo standard IEEE prevede valori che risultano da operazioni non definite, denominati **NaN** (not a number)

- ◆ Un NaN non è uguale a nessun altro numero, nemmeno a sé stesso

```
- val e = Math.sqrt(~2.0);  
val e = nan : real  
  
- Real.==(e,e);  
val it = false : bool
```



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Funzioni

Annotazioni di tipo

Una funzione

ricorsiva

Altre dichiarazioni di
identificatori

Scoping

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplificazione par. funz

Dichiarazioni e scoping in ML



Funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice calcolo di una funzione par. funz

- Ci sono diversi modi di definire e chiamare funzioni in ML.
Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int
```



Funzioni

- Ci sono diversi modi di definire e chiamare funzioni in ML.
Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int  
  
- quadr(3);  
val it = 9 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice calcolatore di polinomi par. funz



Funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice calcolo di una
f1 - soluzione par. funz

- Ci sono diversi modi di definire e chiamare funzioni in ML.
Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int  
  
- quadr(3);  
val it = 9 : int  
  
- quadr 3; (* in questo caso le parentesi sono opzionali *)  
val it = 9 : int
```



Funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice calcolo di combinazioni par. funz

- Ci sono diversi modi di definire e chiamare funzioni in ML.
Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int  
  
- quadr(3);  
val it = 9 : int  
  
- quadr 3; (* in questo caso le parentesi sono opzionali *)  
val it = 9 : int
```

- Con **fun** si *dichiara* la funzione

- ◆ **fun** aggiunge all'ambiente l'identificatore **quadr**
- ◆ e lo associa alla funzione da interi a interi



Funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice calcolo di somma par. funz

- Ci sono diversi modi di definire e chiamare funzioni in ML.
Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int  
  
- quadr(3);  
val it = 9 : int  
  
- quadr 3; (* in questo caso le parentesi sono opzionali *)  
val it = 9 : int
```

- Con **fun** si *dichiara* la funzione
 - ◆ fun aggiunge all'ambiente l'identificatore **quadr**
 - ◆ e lo associa alla funzione da interi a interi
- Si può vedere cosa è associato a **quadr** senza *chiamare la funzione*

```
- quadr; (* nome della funzione senza argomenti *)  
val it = fn : int -> int
```

Mostra solo il tipo (il valore è stato trasformato in bytecode)



Annotazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione

ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice combinatore par. funz

É possibile indicare esplicitamente i tipi:

```
- fun quadr (x :int) : int = x * x;  
val quadr = fn : int -> int
```

In assenza di annotazioni di tipo, scatta la *type inference*



Una funzione ricorsiva

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);  
val fatt = fn : int -> int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice calcolo per funz.



Una funzione ricorsiva

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Funzioni

Annotazioni di tipo

Una funzione
ricorsiva

Altre dichiarazioni di
identificatori

Scoping

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplificazione par. funz

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);  
val fatt = fn : int -> int  
  
- fatt(3);  
val it = 6 : int
```



Una funzione ricorsiva

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Funzioni

Annotazioni di tipo

Una funzione
ricorsiva

Altre dichiarazioni di
identificatori

Scoping

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplificazione par. funz

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);  
val fatt = fn : int -> int  
  
- fatt(3);  
val it = 6 : int  
  
- fatt 3; (* in questo caso le parentesi sono opzionali *)  
val it = 6 : int
```

- In ML il costrutto `if-then-else` denota un'espressione
- In Java/C/C++/etc., denota uno *statement* (non ha un valore)
- Quindi, `if-then-else` in ML è simile all'operatore ternario `?:` in Java/C/C++



Altre dichiarazioni di identificatori

- Con **val** si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore

```
- val x = 2+2;  
val x = 4 : int  
  
- x+2;  
val it = 6 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice ~~combinazione~~ par. funz



Altre dichiarazioni di identificatori

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice ~~combinazione~~ par. funz

- Con **val** si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore

```
- val x = 2+2;  
val x = 4 : int  
  
- x+2;  
val it = 6 : int
```

- Grammatica delle dichiarazioni viste sinora

```
<declaration> ::=  
    val <id name> = <expression> |  
    fun <func name> <argument>* = <expression>
```

Vedremo più avanti che **val** è più generale di **fun**



Scoping

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione

ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice bindazione par. funz

- L'equivalente dei blocchi in ML è

```
let
    <dichiarazioni>
in
    <espressione> (* le dichiarazioni valgono solo qui *)
end
```



Scoping

- L'equivalente dei blocchi in ML è

```
let  
    <dichiarazioni>  
in  
    <espressione> (* le dichiarazioni valgono solo qui *)  
end
```

Lo scoping è **statico**. Esempi:

```
- let val x=2 in 3*x end;  
val it = 6 : int  
  
- x;  
Error: unbound variable or constructor: x  
  
- let val x=2 in  
    let val x=3 in (* questa def. maschera la precedente *)  
        3*x  
    end  
end;  
val it = 9 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione

ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore per funzioni par. funz.



Scoping (II)

■ Ambiente non locale delle funzioni

```
- val x=0;  
val x=0 : int
```

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Funzioni

Annotazioni di tipo

Una funzione
ricorsiva

Altre dichiarazioni di
identificatori

Scoping

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplificazione par. funz



Scoping (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore per funzioni par. funz

Ambiente non locale delle funzioni

```
- val x=0;
val x=0 : int

- let val x=1 in
    let fun f(y) = x+y in      (* x è non locale *)
        f(0)
    end
end;
val it = 1 : int
```



Scoping (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice costruttore per funz.

Ambiente non locale delle funzioni

```
- val x=0;
val x=0 : int

- let val x=1 in
    let fun f(y) = x+y in      (* x è non locale *)
        f(0)
    end
end;
val it = 1 : int
```

Forma equivalente più concisa

```
let
  val x=1
  fun f(y) = x+y
in
  f(0)
end;
```

dopo "let" possiamo mettere quante dichiarazioni vogliamo



Scoping (III)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Funzioni

Annotazioni di tipo

Una funzione

ricorsiva

Altre dichiarazioni di identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice bindazione par. funz

Definizioni ausiliarie

- locali ad altre definizioni

```
local
    <dichiarazioni>
in
    <dichiarazione> (* le dichiarazioni sopra valgono solo qui *)
end
```

Simile a let ma dopo in c'è una dichiarazione invece di una espressione da valutare



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e
costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Tipi strutturati in ML



Prodotti cartesiani

- Si possono definire n -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘*’
- Si estraе l’ i -esimo elemento da una n -upla con l’operatore prefisso $\#i$

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Prodotti cartesiani

- Si possono definire n -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘*’
- Si estraе l’ i -esimo elemento da una n -upla con l’operatore prefisso $\#i$

```
- (1+1, "A");  
val it = (2,"A") : int * string
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Prodotti cartesiani

- Si possono definire n -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘*’
- Si estraе l’ i -esimo elemento da una n -upla con l’operatore prefisso $\#i$

```
- (1+1, "A");
val it = (2,"A") : int * string

- val x = (1,"A",3.5);
val x = (1,"A",3.5) : int * string * real
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Prodotti cartesiani

- Si possono definire n -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘*’
- Si estrae l’ i -esimo elemento da una n -upla con l’operatore prefisso $\#i$

```
- (1+1, "A");
val it = (2,"A") : int * string

- val x = (1,"A",3.5);
val x = (1,"A",3.5) : int * string * real

- #1(x);
val it = 1 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Prodotti cartesiani

- Si possono definire n -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘*’
- Si estraе l’ i -esimo elemento da una n -upla con l’operatore prefisso $\#i$

```
- (1+1, "A");
val it = (2,"A") : int * string

- val x = (1,"A",3.5);
val x = (1,"A",3.5) : int * string * real

- #1(x);
val it = 1 : int

- #2(x);
val it = "A" : string
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Prodotti cartesiani

- Si possono definire n -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘*’
- Si estrae l’ i -esimo elemento da una n -upla con l’operatore prefisso $\#i$

```
- (1+1, "A");
val it = (2,"A") : int * string

- val x = (1,"A",3.5);
val x = (1,"A",3.5) : int * string * real

- #1(x);
val it = 1 : int

- #2(x);
val it = "A" : string

- #3(x);
val it = 3.5 : real
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Record

- Insiemi di espressioni `<nome>=<valore>`. Notate come viene rappresentato il tipo

```
- val r = {nome="Mario",nato=1998};  
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Record

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Insiemi di espressioni `<nome>=<valore>`. Notate come viene rappresentato il tipo

```
- val r = {nome="Mario",nato=1998};  
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```

- Il valore associato al nome N si estrae con `#N`

```
- #nome(r);  
val it = "Mario" : string  
  
- #nato(r);  
val it = 1998 : int
```



Record

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Insiemi di espressioni `<nome>=<valore>`. Notate come viene rappresentato il tipo

```
- val r = {nome="Mario",nato=1998};  
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```

- Il valore associato al nome N si estrae con `#N`

```
- #nome(r);  
val it = "Mario" : string  
  
- #nato(r);  
val it = 1998 : int
```

- L'ordine delle coppie non conta

```
- {nome="Mario", nato=1998} = {nato=1998, nome="Mario"};  
val it = true : bool
```



Dichiarazioni di tipo

- ML permette di definire nuovi tipi similmente ai `typedef` del C

```
- type coord = real * real;  
type coord = real * real
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai `typedef` del C

```
- type coord = real * real;  
type coord = real * real
```

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0); (* senza aiutino *)  
val x = (3.0,4.0) : real * real  
  
- val x:coord = (3.0,4.0); (* con aiutino *)  
val x = (3.0,4.0) : coord
```

- I tipi `coord` e `(real * real)` sono compatibili tra loro (*structural equivalence*)



Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai `typedef` del C

```
- type coord = real * real;  
type coord = real * real
```

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0); (* senza aiutino *)  
val x = (3.0,4.0) : real * real  
  
- val x:coord = (3.0,4.0); (* con aiutino *)  
val x = (3.0,4.0) : coord
```

- I tipi `coord` e `(real * real)` sono compatibili tra loro (*structural equivalence*)

- ◆ posso passare una espressione di tipo `coord` a un parametro di tipo `(real * real)` e viceversa



Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai `typedef` del C

```
- type coord = real * real;  
type coord = real * real
```

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0); (* senza aiutino *)  
val x = (3.0,4.0) : real * real  
  
- val x:coord = (3.0,4.0); (* con aiutino *)  
val x = (3.0,4.0) : coord
```

- I tipi `coord` e `(real * real)` sono compatibili tra loro (*structural equivalence*)

- ◆ posso passare una espressione di tipo `coord` a un parametro di tipo `(real * real)` e viceversa
- ◆ similmente `coord` è compatibile con ogni altro tipo definito come `(real * real)`, come ad esempio

```
type coppia = real * real;
```



Datatypes e costruttori

■ Con **datatype** si può fare di più che dare un nome a un tipo ML

◆ si possono definire **costruttori** per creare *data objects*

```
- datatype color = red | green | blue;  
datatype color = blue | green | red
```

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e
costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore



Datatypes e costruttori

- Con **datatype** si può fare di più che dare un nome a un tipo ML

- ◆ si possono definire **costruttori** per creare *data objects*

```
- datatype color = red | green | blue;  
datatype color = blue | green | red  
  
- val c = red;  
val c = red : color
```

red, green, blue sono costruttori. Definiscono i possibili valori del tipo color

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Datatypes e costruttori

- Con **datatype** si può fare di più che dare un nome a un tipo ML

- ◆ si possono definire **costruttori** per creare *data objects*

```
- datatype color = red | green | blue;  
datatype color = blue | green | red  
  
- val c = red;  
val c = red : color
```

red, green, blue sono costruttori. Definiscono i possibili valori del tipo color

- Notare la somiglianza con le enum del C. Solo apparente...

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d", red, green, blue);      (* stampa 012 *)
```



Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d", red, green, blue);      (* stampa 012 *)

enum color c = red;
if( c == 0 ) then ... else ...;          (* esegue il then *)
```



Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d", red, green, blue);      (* stampa 012 *)

enum color c = red;
if( c == 0 ) then ... else ...;          (* esegue il then *)

c = 10;                                  (* nessun errore!!! *)
```



Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d", red, green, blue);      (* stampa 012 *)

enum color c = red;
if( c == 0 ) then ... else ...;          (* esegue il then *)

c = 10;                                  (* nessun errore!!! *)
```

- Invece i datatypes di ML definiscono tipi genuinamente nuovi: nessuna corrispondenza con gli int

```
- val c:color = 10;
Error: pattern and expression in val dec don't agree

- c = 0;                                (* c è uguale a 0? *)
Error: operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi



Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };
printf("%d%d%d", red, green, blue);      (* stampa 012 *)

enum color c = red;
if( c == 0 ) then ... else ...;          (* esegue il then *)

c = 10;                                  (* nessun errore!!! *)
```

- Invece i datatypes di ML definiscono tipi genuinamente nuovi: nessuna corrispondenza con gli int

```
- val c:color = 10;
Error: pattern and expression in val dec don't agree

- c = 0;                                (* c è uguale a 0? *)
Error: operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi

- Ogni tipo definito con datatype è incompatibile con *tutti gli altri tipi (name equivalence)*



Costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani
Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
 - ◆ la lista vuota (caso base)
 - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)



Costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

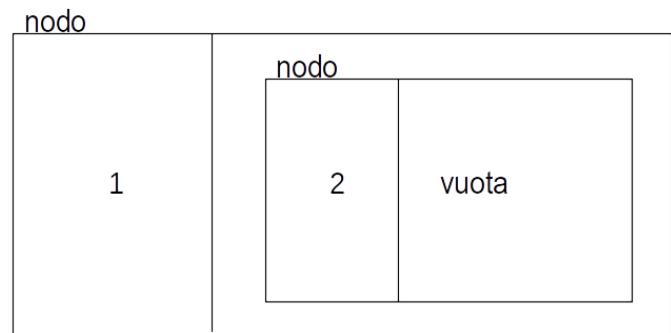
Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
 - ◆ la lista vuota (caso base)
 - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)





Costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

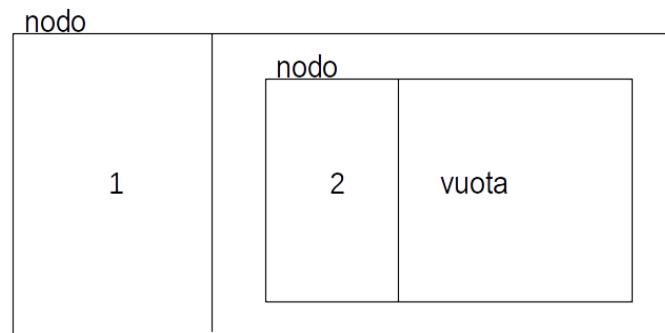
Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
 - ◆ la lista vuota (caso base)
 - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)



Rappresentazione della lista [1,2]

- Quindi servono 2 costruttori: per la lista vuota e per i nodi

```
- datatype listaInt = vuota | nodo of int * listaInt;  
datatype listaInt = nodo of int * listaInt | vuota
```



Costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

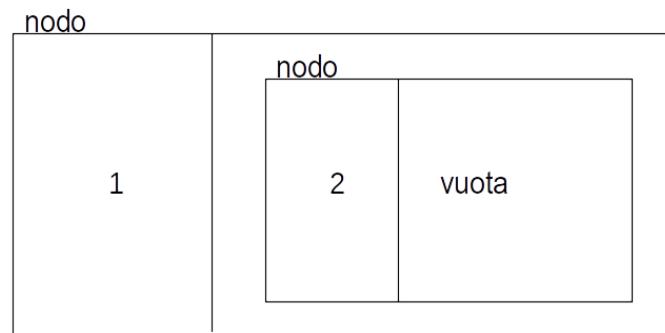
Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
 - ◆ la lista vuota (caso base)
 - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)



Rappresentazione della lista [1,2]

- Quindi servono 2 costruttori: per la lista vuota e per i nodi

```
- datatype listaInt = vuota | nodo of int * listaInt;
datatype listaInt = nodo of int * listaInt | vuota

- val L = nodo(1, nodo(2, vuota));
val L = nodo (1,nodo (2,vuota)) : listaInt
```



Costruttori con argomenti (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Altro esempio: albero binario con nodi etichettati da interi

- Definizione ricorsiva: un albero simile è
 - ◆ un albero vuoto, oppure
 - ◆ un nodo che contiene un intero e due alberi dello stesso tipo



Costruttori con argomenti (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Altro esempio: albero binario con nodi etichettati da interi

■ Definizione ricorsiva: un albero simile è

- ◆ un albero vuoto, oppure
- ◆ un nodo che contiene un intero e due alberi dello stesso tipo

```
datatype albero = vuoto | nodoAlb of int * albero * albero
```



Costruttori con argomenti (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Altro esempio: albero binario con nodi etichettati da interi

■ Definizione ricorsiva: un albero simile è

- ◆ un albero vuoto, oppure
- ◆ un nodo che contiene un intero e due alberi dello stesso tipo

```
datatype albero = vuoto | nodoAlb of int * albero * albero
```

In questo esempio costruiamo un albero con radice 1, figlio sinistro 2 (che è una foglia), mentre il figlio destro manca.

```
nodoAlb (1, nodoAlb (2, vuoto, vuoto), vuoto)
```



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Patterns e matching



Utilizzo dei costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

```
- val L = nodo(1, nodo(2, vuota));  
val L = nodo (1,nodo (2,vuota)) : listaInt  
  
- L = vuota;  
val it = false : bool
```



Utilizzo dei costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

```
- val L = nodo(1, nodo(2, vuota));  
val L = nodo (1,nodo (2,vuota)) : listaInt  
  
- L = vuota;  
val it = false : bool
```

- Se non è vuota potrebbe servirci il primo elemento. Si estrae con *pattern matching*

```
- val nodo(p,_) = L;      (* assegna a p il 1° elemento di L *)  
val p = 1 : int           (* "_" è una wildcard *)
```

La parte in rosso è chiamata *pattern*



Utilizzo dei costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

```
- val L = nodo(1, nodo(2, vuota));  
val L = nodo (1,nodo (2,vuota)) : listaInt  
  
- L = vuota;  
val it = false : bool
```

- Se non è vuota potrebbe servirci il primo elemento. Si estrae con *pattern matching*

```
- val nodo(p,_) = L;      (* assegna a p il 1° elemento di L *)  
val p = 1 : int           (* "_" è una wildcard *)
```

La parte in rosso è chiamata *pattern*

- Per ottenere il resto della lista

```
- val nodo(_,r) = L;          (* assegna a r il resto *)  
val r = nodo (2,vuota) : listaInt
```



Funzione che conta gli elementi della lista

- Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x =
    if x = vuota then 0
        else
            let val nodo(_, r) = x in
                conta(r) + 1
            end;
val conta = fn : listaInt -> int

- conta L;
val it = 2 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Funzione che conta gli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x =
    if x = vuota then 0
    else
        let val nodo(_, r) = x in
            conta(r) + 1
        end;
val conta = fn : listaInt -> int

- conta L;
val it = 2 : int
```

- Notare come il compilatore ha *inferito* il tipo della funzione
 1. x viene confrontato con “vuota”, che è di tipo listaInt \Rightarrow anche x è di tipo listaInt \Rightarrow l’input di “conta” è un listaInt
 2. il “then” restituisce 0, che è un intero; quindi l’output di “conta” è un intero



Funzione che conta gli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x =
    if x = vuota then 0
    else
        let val nodo(_, r) = x in
            conta(r) + 1
        end;
val conta = fn : listaInt -> int

- conta L;
val it = 2 : int
```

- Inoltre il compilatore controlla che anche il resto della funzione sia compatibile con questi tipi
 1. r corrisponde al 2° argomento del nodo, che è di tipo listaInt \Rightarrow è corretto passarlo a conta che restituisce un intero
 2. quindi anche l'else restituisce un intero, e tutto torna



Abbreviazioni per i pattern

- Si possono estrarre tutti gli elementi di un costruttore in un colpo solo

```
- val nodo(p, r) = L;  
val p = 1 : int  
val r = nodo (2, vuota) : listaInt
```

Cioè dichiara due identificatori (p e r) in un colpo solo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Abbreviazioni per i pattern

- Si possono estrarre tutti gli elementi di un costruttore in un colpo solo

```
- val nodo(p, r) = L;  
val p = 1 : int  
val r = nodo (2, vuota) : listaInt
```

Cioè dichiara due identificatori (p e r) in un colpo solo

- Si può *definire una funzione per casi*

```
- fun conta(vuota)      = 0  
    | conta(nodo(_, r)) = conta(r) + 1;
```

mettendo direttamente i pattern al posto dei parametri formali

Notare l'eleganza e la concisione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Analogo per i pattern dello switch/case del C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Def. per casi

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- L'idea è la stessa della definizione per casi delle funzioni

```
- case L of vuota => true  
| nodo(_, _) => false;  
  
val it = false : bool
```



Esercizi

1. Scrivere in ML una funzione `size` che restituisce il *numero di nodi* in un albero binario come quello visto in precedenza
2. Scrivere in ML una funzione che, dato un albero binario A come quello visto in precedenza, e un intero N, restituisce l'etichetta del nodo che si raggiunge in N passi visitando l'albero in preordine
3. Simile all'esercizio 2, ma visitando l'albero in postordine (difficile)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

Patterns

Def. per casi

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Liste



Le liste in ML

- Le liste sono tra le strutture dati più usate in programmazione funzionale
 - ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Le liste in ML

- Le liste sono tra le strutture dati più usate in programmazione funzionale

- ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)

- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

```
nil                      (* lista vuota *)
1 :: 2 :: 3 :: nil      (* lista che contiene 1, 2, 3 *)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Le liste in ML

- Le liste sono tra le strutture dati più usate in programmazione funzionale

- ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)

- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

```
nil                      (* lista vuota *)
1 :: 2 :: 3 :: nil      (* lista che contiene 1, 2, 3 *)

(* formato equivalente basato su parentesi quadre *)
[]
[1,2,3]
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Le liste in ML

- Le liste sono tra le strutture dati più usate in programmazione funzionale

- ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)

- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

```
nil                      (* lista vuota *)
1 :: 2 :: 3 :: nil      (* lista che contiene 1, 2, 3 *)

(* formato equivalente basato su parentesi quadre *)
[]
[1,2,3]

(* sono veramente equivalenti *)
- [] = nil;
val it = true : bool

- 1::2::3::nil = [1,2,3];
val it = true : bool
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Principali operatori sulle liste in ML

- La funzione `length` restituisce la lunghezza di una lista

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Principali operatori sulle liste in ML

- La funzione `length` restituisce la lunghezza di una lista
- La funzione `null` restituisce true se la lista è vuota

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore



Principali operatori sulle liste in ML

- La funzione `length` restituisce la lunghezza di una lista
- La funzione `null` restituisce true se la lista è vuota
- Le funzioni `hd` (*head*) e `tl` (*tail*) restituiscono il primo elemento e il resto della lista, rispettivamente

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Principali operatori sulle liste in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La funzione `length` restituisce la lunghezza di una lista
- La funzione `null` restituisce true se la lista è vuota
- Le funzioni `hd` (*head*) e `tl` (*tail*) restituiscono il primo elemento e il resto della lista, rispettivamente
- L'operatore infisso `@` restituisce la concatenazione di due liste

```
- val L = [1,2,3];
val L = [1,2,3] : int list

- hd L;
val it = 1 : int

- tl L;
val it = [2,3] : int list

- val M = [1,2] @ [3,4];
val M = [1,2,3,4] : int list
```



Funzioni aggiuntive su liste

- Altre funzioni si trovano nella struttura List, ad esempio
 - ◆ `List.nth(L,i)` restituisce l' i -esimo elemento di L (partendo da 0)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Funzioni aggiuntive su liste

- Altre funzioni si trovano nella struttura List, ad esempio
 - ◆ `List.nth(L,i)` restituisce l' i -esimo elemento di L (partendo da 0)
 - ◆ `List.last(L)` restituisce l'ultimo elemento di L
- vedere <http://sml-family.org/Basis/list.html>
- In ML, una *struttura* è simile a un modulo o un package

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore



Esercizi

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Scrivere in ML:

1. le funzioni standard su liste riportate nelle slide precedenti: `hd`, `tl`, `List.nth`, `List.last`;
2. una funzione `member` che dati una lista `L` e un intero `N`, restituisce `true` se e solo se `N` appartiene ad `L`.
3. una funzione `append` che, dati una lista `L` e un intero `N`, accoda `N` in fondo alla lista (inserimento in coda);
4. una funzione `concat` che concatena due liste (come l'operatore `@`);
5. una funzione `reverse` che inverte l'ordine degli elementi di una lista (suggerimento: usare un parametro aggiuntivo che serve a costruire progressivamente la lista invertita).



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Currying



Funzioni con più argomenti: esistono?

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- In realtà in ML ogni funzione ha un solo argomento

```
fun f (x,y) = ...      (* l'argomento è una (singola) coppia *)
```

```
fun f x y = ...      (* l'argomento è x ! *)
```

Nel secondo caso, *f* è una funzione che *restituisce una funzione* che prende *y* e calcola l'espressione dopo '='



Funzioni con più argomenti: esistono?

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- In realtà in ML ogni funzione ha un solo argomento

```
fun f (x,y) = ... (* l'argomento è una (singola) coppia *)
```

```
fun f x y = ... (* l'argomento è x ! *)
```

Nel secondo caso, *f* è una funzione che *restituisce una funzione* che prende *y* e calcola l'espressione dopo '='

- Esempio semplice

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int
```

Il tipo $\text{int} \rightarrow \text{int} \rightarrow \text{int}$ va inteso come $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$



Currying

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int
```



Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int
```



Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;          (* viene interpretato come (f'3)(2) *)  
val it = 5 : int
```



Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;          (* viene interpretato come (f'3)(2) *)  
val it = 5 : int  
  
- f 3 2;  
Error: operator and operand don't agree
```



Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;          (* viene interpretato come (f'3)(2) *)  
val it = 5 : int  
  
- f 3 2;  
Error: operator and operand don't agree  
  
- f' (3,2);  
Error: operator and operand don't agree
```



Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;          (* viene interpretato come (f'3)(2) *)  
val it = 5 : int  
  
- f 3 2;  
Error: operator and operand don't agree  
  
- f' (3,2);  
Error: operator and operand don't agree  
  
- val g = f' 3;  
val g = fn : int -> int
```



Currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La trasformazione da n-uple (come $f(x,y)$) a funzioni che restituiscono funzioni (come $f' x y$) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;          (* viene interpretato come (f'3)(2) *)  
val it = 5 : int  
  
- f 3 2;  
Error: operator and operand don't agree  
  
- f' (3,2);  
Error: operator and operand don't agree  
  
- val g = f' 3;  
val g = fn : int -> int  
  
- g 1;  
val it = 4 : int
```



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Funzioni di ordine superiore



Invece dei cicli: Funzioni di ordine superiore

- La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura
- Cambia solo l'operazione che si applica ai nodi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Invece dei cicli: Funzioni di ordine superiore

- La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura
- Cambia solo l'operazione che si applica ai nodi
- Quindi basta scrivere una volta per tutte la funzione che scandisce la struttura dati (che fa la funzione del ciclo)
- e passargli la funzione da applicare ai nodi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Invece dei cicli: Funzioni di ordine superiore

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura
- Cambia solo l'operazione che si applica ai nodi
- Quindi basta scrivere una volta per tutte la funzione che scandisce la struttura dati (che fa la funzione del ciclo) e passargli la funzione da applicare ai nodi
 - ◆ Le funzioni che hanno altre funzioni come parametri sono dette di *ordine superiore*



Esempio

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Una funzione f che esegue due volte una funzione g su un valore x:

```
- fun f g x = g(g(x));  
val f = fn : ('a -> 'a) -> 'a -> 'a
```

Equivalente in C (solo per interi):

```
int f( int (*g)(int), int x) {  
    return g(g(x));  
}
```



Invece dei cicli: Funzioni di ordine superiore

■ Le tipologie di funzioni/ciclo più comuni sono tre:

- ◆ filter
- ◆ map
- ◆ reduce/fold

Nel seguito mostriamo le loro versioni per le liste

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Filter

- La funzione filter prende una funzione booleana f e una lista L
- seleziona gli elementi di L per cui f è vera

```
fun filter f [] = []
| filter f (x::y) = if f(x) then x::(filter f y)
                     else filter f y
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Filter

- La funzione filter prende una funzione booleana f e una lista L
- seleziona gli elementi di L per cui f è vera

```
fun filter f [] = []
| filter f (x::y) = if f(x) then x::(filter f y)
                     else filter f y

(* esempio: seleziona gli elementi negativi da una lista *)
- let fun neg x = x<0
      in filter neg [0,~1,3,~2] end;
val it = [~1,~2] : int list
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Filter

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La funzione filter prende una funzione booleana f e una lista L
- seleziona gli elementi di L per cui f è vera

```
fun filter f [] = []
  | filter f (x::y) = if f(x) then x::(filter f y)
                      else filter f y

(* esempio: seleziona gli elementi negativi da una lista *)
- let fun neg x = x<0
    in filter neg [0,~1,3,~2] end;
val it = [~1,~2] : int list

(* esempio: seleziona gli elementi positivi da una lista *)
- let fun pos x = x>0
    in filter pos [0,~1,3,~2] end;
val it = [3] : int list
```

Nella libreria standard: List.filter



Map

- La funzione `map` prende una funzione `f` e una lista `L`
- applica `f` a tutti gli elementi della lista, ottenendo una nuova lista

```
fun map f [] = []
| map f (x::y) = f(x)::(map f y)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter,Map,Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Map

- La funzione `map` prende una funzione `f` e una lista `L`
- applica `f` a tutti gli elementi della lista, ottenendo una nuova lista

```
fun map f [] = []
| map f (x::y) = f(x)::(map f y)

(* esempio: conversione in lista di reali *)
- map real [1,2,3];
val it = [1.0,2.0,3.0] : real list
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Map

- La funzione `map` prende una funzione `f` e una lista `L`
- applica `f` a tutti gli elementi della lista, ottenendo una nuova lista

```
fun map f [] = []
| map f (x::y) = f(x)::(map f y)

(* esempio: conversione in lista di reali *)
- map real [1,2,3];
val it = [1.0,2.0,3.0] : real list

(* esempio: conversione in lista di stringhe *)
- map Int.toString [1,2,3];
val it = ["1","2","3"] : string list
```

Nella libreria standard: `List.map`

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Reduce/fold

- La funzione `reduce` serve per calcolare *aggregati* di una lista
 - ◆ min, max, somma, prodotto, media...

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter,Map,Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

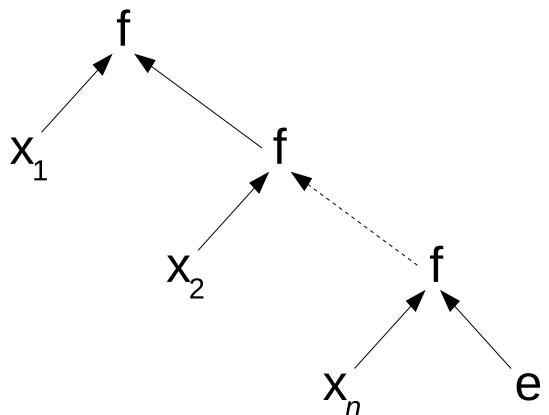
[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Reduce/fold

- La funzione `reduce` serve per calcolare *aggregati* di una lista
 - ◆ min, max, somma, prodotto, media...
- Prende in input una funzione a 2 argomenti f , un valore finale e e una lista L ed effettua questo calcolo:

$$\text{reduce } f \text{ e } [x_1, x_2, \dots, x_n] = f(x_1, f(x_2, \dots, f(x_n, e) \dots))$$


Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo
parametrico

Encapsulation e
interfacce

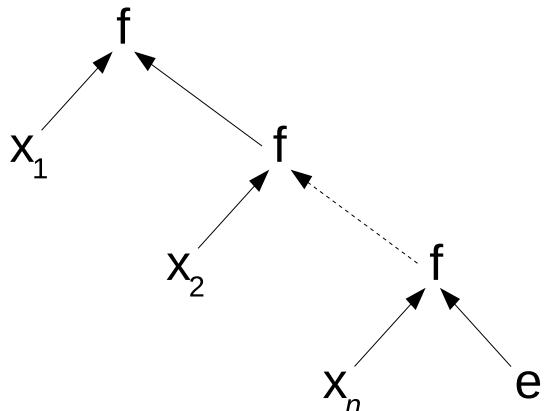
Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore



Reduce/fold

- La funzione `reduce` serve per calcolare *aggregati* di una lista
 - ◆ min, max, somma, prodotto, media...
- Prende in input una funzione a 2 argomenti f , un valore finale e e una lista L ed effettua questo calcolo:

$$\text{reduce } f \text{ e } [x_1, x_2, \dots, x_n] = f(x_1, f(x_2, \dots, f(x_n, e) \dots))$$


Ad esempio se f è '+' ed $e=0$ allora `reduce` calcola la somma degli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Reduce

■ Definizione di reduce ed esempi

```
fun reduce f e [] = e
| reduce f e (x :: y) = f (x, reduce f e y)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter,Map,Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Reduce

■ Definizione di reduce ed esempi

```
fun reduce f e [] = e
| reduce f e (x :: y) = f (x, reduce f e y)

(* esempio: somma (sbagliato, + è infisso...) *)
- reduce + 0 [1,2,3];
Error: ...

(* esempio: somma (corretto) *)
- reduce (op +) 0 [1,2,3];
val it = 6 : int
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Reduce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Definizione di reduce ed esempi

```
fun reduce f e [] = e
| reduce f e (x :: y) = f (x, reduce f e y)

(* esempio: somma (sbagliato, + è infisso...) *)
- reduce + 0 [1,2,3];
Error: ...

(* esempio: somma (corretto) *)
- reduce (op +) 0 [1,2,3];
val it = 6 : int

(* esempio: media di [x1,..,xn] = x1/n + ... + xn/n *)
- let
    fun f L (elem,accum) = elem / real(length L) + accum
    val lista = [1.0,2.0,3.0]
  in
    reduce (f lista) 0.0 lista
  end;
val it = 2.0 : real
```



Reduce

- Simile alla funzione standard `List.foldr`

- `List.foldr f e l` restituisce:

$f(x_1, f(x_2, \dots, f(x_{n-1}, f(x_n, e))))$

- Su lista vuota, restituisce e

- È più generale di reduce, perché f può accettare due tipi diversi:

$f : 'a * 'b \rightarrow 'b$

- Tipo di `List.foldr`:

$fn : ('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a list \rightarrow 'b$

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore



Esempio di foldr

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
- fun f (n:int, s:string) = Int.toString(n) ^ ";" ^ s;
val f = fn : int * string -> string

- f (323, "aaa");
val it = "323;" ^ "aaa" : string

- List.foldr f "fine" [23,3434,44,0,12];
val it = "23;" ^ "3434;" ^ "44;" ^ "0;" ^ "12;" ^ "fine" : string
```



In altri linguaggi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

In Java 8+ (semplificato):

```
interface java.util.stream.Stream<T> {  
    ...  
    Stream<T> filter(Predicate<T> predicate)  
    <R> Stream<R> map(Function<T,R> mapper)  
    T reduce(T identity, BinaryOperator<T> accumulator)  
}
```

In Python:

```
filter(funzione lista)  
map(funzione, lista)  
reduce(funzione, lista [, inizializzatore])
```



Esercizi su funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Implementare in ML le seguenti funzioni, sfruttando al meglio filter, map e reduce

1. pari: data una lista di interi L, restituisce la sottolista che contiene solo i numeri pari in L
2. membro: dati una lista di interi e un intero, restituisce *true* se e solo se l'intero compare nella lista
3. concat: data una lista di stringhe, restituisce la loro concatenazione
4. max: data una lista di interi, restituisce il massimo



Funzioni anonymous

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonymous

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Quando utilizziamo funzioni di ordine superiori come filter, map e reduce, può far comodo passargli funzioni semplici, specificate lì per lì senza dover dare loro un nome (né usare un blocco *let*)
- Queste funzioni anonymous si specificano con la keyword **fn**

Sintassi:

```
fn <argomento> => <espressione>
```

Esempi:

```
- fn x => x+1;  
val it = fn : int -> int  
  
(* per sommare uno a tutti gli elementi di una lista *)  
- map (fn x => x+1) [1,2,3];  
val it = [2,3,4] : int list  
  
- filter (fn x => (x mod 2) = 0) [10,11,12,13];  
val it = [10,12] : int list
```



In altri linguaggi funzionali

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

In Lisp e Scheme l'equivalente di `fn` è la keyword *lambda*

- storicamente deriva dal *lambda calcolo*, un modello di calcolo matematico basato su funzioni di ordine superiore a cui tutti i linguaggi funzionali si sono ispirati
- nel lambda calcolo l'operatore λ è l'analogo di `fn`
 - ◆ come in $\lambda x. x + 1$



In altri linguaggi imperativi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Una funzione anonima che accetta una stringa e restituisce vero se è più lunga di 10 caratteri

In Java:

```
s -> s.length() > 10;
```

In C++:

```
[](string s) { return s.length() > 10; };
```

In Python:

```
lambda s: len(s) > 10
```



Esercizi su funzioni anonime

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter,Map,Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

1. Spiegare perchè una funzione anonima non può essere ricorsiva
2. Svolgere nuovamente gli esercizi sulle funzioni di ordine superiore, sfruttando al meglio le *funzioni anonime*



Val vs. fun

- Adesso possiamo vedere in che senso val è più generale di fun. Le seguenti definizioni sono equivalenti:

```
- fun f x = x + 1;  
val f = fn : int -> int  
  
- val f = fn x => x+1;  
val f = fn : int -> int
```

In altre parole, fun è *zucchero sintattico*, cioè una utile abbreviazione per qualcosa che si potrebbe fare in altro modo (con val)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Ulteriori dettagli su currying

- Adesso possiamo mostrare più esplicitamente la natura del currying. Riprendiamo l'esempio

```
- fun f' x y = x+y;
```

- In effetti f' può essere definita equivalentemente come

```
fun f' x = fn y => x+y
```

```
val f' = fn : int -> int -> int
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter,Map,Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Ulteriori dettagli su currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Adesso possiamo mostrare più esplicitamente la natura del currying. Riprendiamo l'esempio

```
- fun f' x y = x+y;
```

- In effetti f' può essere definita equivalentemente come

```
fun f' x = fn y => x+y
```

```
val f' = fn : int -> int -> int
```

- L'esempio della chiamata di f' con un solo parametro può essere spiegata così:

```
- val g = f' 3;          (* come fosse val g = fn y => 3+y *)  
val g = fn : int -> int
```

```
- g 1;  
val it = 4 : int
```



Ulteriori dettagli su currying

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)
val curry_2args = fn f => fn x => fn y => f (x, y)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Ulteriori dettagli su currying

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)
val curry_2args = fn f => fn x => fn y => f (x, y)

(* f deve essere del tipo f x y *)
val uncurry_2args = fn f => fn (x, y) => f x y
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Ulteriori dettagli su currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce
Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)
val curry_2args = fn f => fn x => fn y => f (x, y)

(* f deve essere del tipo f x y *)
val uncurry_2args = fn f => fn (x, y) => f x y
```

- Esempi di utilizzo

```
fun f (x,y) = x+y;

val f' = curry_2args f; (* equivalente a f' x y = x+y *)
```



Ulteriori dettagli su currying

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter,Map,Reduce
Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)
val curry_2args = fn f => fn x => fn y => f (x, y)

(* f deve essere del tipo f x y *)
val uncurry_2args = fn f => fn (x, y) => f x y
```

- Esempi di utilizzo

```
fun f (x,y) = x+y;

val f' = curry_2args f; (* equivalente a f' x y = x+y *)
```

oppure

```
fun f' x y = x+y;

val f = uncurry_2args f'; (* equivalente a f(x,y) = x+y *)
```



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Tipi parametrici](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Polimorfismo parametrico



Tipi parametrici

- ML supporta l'analogo dei *template* di C++ e Java, ovvero *tipi parametrici*

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

Tipi parametrici

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Tipi parametrici

- ML supporta l'analogo dei *template* di C++ e Java, ovvero *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
 - ◆ il costruttore :: può essere applicato a qualunque tipo

```
1 :: nil    oppure    "abc" :: nil ...
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML supporta l'analogo dei *template* di C++ e Java, ovvero *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
 - ◆ il costruttore :: può essere applicato a qualunque tipo

```
1 :: nil    oppure    "abc" :: nil    ...
- length;
val it = fn : 'a list -> int
```

La funzione *length* prende una lista di elementi il cui tipo '*a*' non è specificato

- ◆ cioè accetta liste con qualsiasi contenuto
- ◆ in effetti non ha bisogno di saperne il tipo: deve solo contare i nodi



Tipi parametrici

- ML supporta l'analogo dei *template* di C++ e Java, ovvero *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
 - ◆ il costruttore :: può essere applicato a qualunque tipo

```
1 :: nil    oppure    "abc" :: nil    ...
- length;
val it = fn : 'a list -> int
```

La funzione *length* prende una lista di elementi il cui tipo '*a* non è specificato

- ◆ cioè accetta liste con qualsiasi contenuto
- ◆ in effetti non ha bisogno di saperne il tipo: deve solo contare in nodi
- Anche la funzione *map* è parametrica:

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```



Tipi parametrici

■ Come definire tipi parametrici (come le liste)

```
(* generalizzazione delle nostre liste *)
```

```
- datatype 'a lista = vuota | nodo of ('a * 'a lista);  
datatype 'a lista = nodo of 'a * 'a lista | vuota
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

Tipi parametrici

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Come definire tipi parametrici (come le liste)

```
(* generalizzazione delle nostre liste *)
```

```
- datatype 'a lista = vuota | nodo of ('a * 'a lista);  
datatype 'a lista = nodo of 'a * 'a lista | vuota
```

```
(* alberi binari con etichette parametriche *)
```

```
- datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);  
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```



Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Come definire tipi parametrici (come le liste)

```
(* generalizzazione delle nostre liste *)
```

```
- datatype 'a lista = vuota | nodo of ('a * 'a lista);  
datatype 'a lista = nodo of 'a * 'a lista | vuota
```

```
(* alberi binari con etichette parametriche *)
```

```
- datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);  
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```

■ Per le funzioni (quando il compilatore non ha bisogno di aiuto per stabilirne il tipo) non dobbiamo fare niente di speciale

◆ ci pensa la *type inference*

```
- fun conta(vuota) = 0  
    | conta (nodo(x,l)) = conta(l) + 1;  
val conta = fn : 'a lista -> int
```



Tipi parametrici

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
 - ◆ ad esempio '*a* o '*b* o '*c* ...

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

Tipi parametrici

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Tipi parametrici

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
 - ◆ ad esempio '*a* o '*b* o '*c* ...
- Quando si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice
 - ◆ ad esempio "*a* o "*b* o "*c* ...

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

Tipi parametrici

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
 - ◆ ad esempio '**a** o '**b** o '**c** ...
- Quando si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice
 - ◆ ad esempio "**a** o "**b** o "**c** ...
- Ogni tanto la type inference se ne accorge da sola

```
- fun diag (x,y) = x=y;
val diag = fn : "a * "a -> bool

- diag (1.0, 1.0);
Error: operator and operand don't agree [equality type required]
      operator domain: ''Z * ''Z
      operand:           real * real
```

(ricordarsi che i reali non supportano l'uguaglianza...)



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[**Encapsulation e interfacce**](#)

[Signatures](#)

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Encapsulation e interfacce



Signatures = Interfacce

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =
sig
    type 'a stack
    val empty: 'a stack
    val push: ('a * 'a stack) -> 'a stack
    val pop: 'a stack -> ('a * 'a stack)
end;
```



Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures
Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =
sig
  type 'a stack
  val empty: 'a stack
  val push: ('a * 'a stack) -> 'a stack
  val pop: 'a stack -> ('a * 'a stack)
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito



Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =
sig
    type 'a stack
    val empty: 'a stack
    val push: ('a * 'a stack) -> 'a stack
    val pop: 'a stack -> ('a * 'a stack)
end;
```

- dichiara un tipo parametrico `'a stack` senza dire com'è definito
- una funzione `empty` per costruire uno stack vuoto



Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =
sig
  type 'a stack
  val empty: 'a stack
  val push: ('a * 'a stack) -> 'a stack
  val pop: 'a stack -> ('a * 'a stack)
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito
- una funzione empty per costruire uno stack vuoto
- una funzione push per inserire un elemento nello stack



Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =
sig
    type 'a stack
    val empty: 'a stack
    val push: ('a * 'a stack) -> 'a stack
    val pop: 'a stack -> ('a * 'a stack)
end;
```

- dichiara un tipo parametrico `'a stack` senza dire com'è definito
- una funzione `empty` per costruire uno stack vuoto
- una funzione `push` per inserire un elemento nello stack
- una funzione `pop` per estrarre la testa dallo stack
- ...senza dire come sono implementate (ovviamente)...



Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =
sig
    type 'a stack
    val empty: 'a stack
    val push: ('a * 'a stack) -> 'a stack
    val pop: 'a stack -> ('a * 'a stack)
end;
```

- dichiara un tipo parametrico `'a stack` senza dire com'è definito
- una funzione `empty` per costruire uno stack vuoto
- una funzione `push` per inserire un elemento nello stack
- una funzione `pop` per estrarre la testa dallo stack
- ...senza dire come sono implementate (ovviamente)...
- *per assegnare un tipo a una espressione usare :*



Structures = Implementazioni delle signature

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =
  struct
    type 'a stack = 'a list;
    val empty = [];
    fun push (x,s) = x :: s;
    fun pop (x::s) = (x,s);
  end;
```



Structures = Implementazioni delle signature

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =
  struct
    type 'a stack = 'a list;
    val empty = [];
    fun push (x,s) = x :: s;
    fun pop (x::s) = (x,s);
  end;
```

- Con l'espressione Stack :> STACK diciamo diverse cose:
 1. Stack deve implementare tutti gli identificatori dichiarati in STACK



Structures = Implementazioni delle signature

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =
  struct
    type 'a stack = 'a list;
    val empty = [];
    fun push (x,s) = x :: s;
    fun pop (x::s) = (x,s);
  end;
```

- Con l'espressione Stack :> STACK diciamo diverse cose:
 1. Stack deve implementare tutti gli identificatori dichiarati in STACK
 2. I tipi di dato dichiarati in Stack possono essere utilizzati *solo* con le operazioni dichiarate in STACK
 - ◆ ogni altra funzione definita nella structure non è accessibile da fuori



Structures = Implementazioni delle signature

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =
  struct
    type 'a stack = 'a list;
    val empty = [];
    fun push (x,s) = x :: s;
    fun pop (x::s) = (x,s);
  end;
```

- Con l'espressione Stack :> STACK diciamo diverse cose:
 1. Stack deve implementare tutti gli identificatori dichiarati in STACK
 2. I tipi di dato dichiarati in Stack possono essere utilizzati *solo* con le operazioni dichiarate in STACK
 - ◆ ogni altra funzione definita nella structure non è accessibile da fuori
 - ◆ così si ottiene l'*encapsulation*
 - ◆ e si definiscono *tipi di dato astratti*



Incapsulamento dell'implementazione dei tipi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Anche se in Stack il tipo stack è implementato con list...

```
type 'a stack = 'a list;
```

- ... non si può usare come list ...



Incapsulamento dell'implementazione dei tipi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Anche se in Stack il tipo stack è implementato con list...

```
type 'a stack = 'a list;
```

- ... non si può usare come list ...

```
- length [];
  val it = 0 : int

- length Stack.empty;
  stdIn:39.1-39.19 Error: operator and operand don't agree
    operator domain: 'Z list
    operand:           'Y Stack.stack
```

- ... perchè la structure Stack non mette a disposizione alcuna funzione length sul tipo stack e ne nasconde l'implementazione



Functors = structure parametriche

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: functor. Analogo dei template

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

[Structures](#)

[Incapsulamento](#)

Functors

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Functors = structure parametriche

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: `functor`. Analogo dei template
- Ecco un esempio di definizione di immagini parametrica rispetto alla codifica del colore
 - ◆ supponiamo di avere due structure RGB e CMYK per gli omonimi modelli di colore
 - ◆ e che entrambe implementino la signature COLOR
 - ◆ la struttura parametrica si può dichiarare così:



Functors = structure parametriche

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: **functor**. Analogo dei template
- Ecco un esempio di definizione di immagini parametrica rispetto alla codifica del colore
 - ◆ supponiamo di avere due structure RGB e CMYK per gli omonimi modelli di colore
 - ◆ e che entrambe implementino la signature COLOR
 - ◆ la struttura parametrica si può dichiarare così:

```
functor Image( X : COLOR ) =
  struct
    (* qui si può usare X come un tipo *)
    (* con le operazioni definite da COLOR *)
  end

  (* col functor si possono generare diversi tipi di dato *)
  structure Image_RGB   = Image(RGB);
  structure Image_CMYK = Image(CMYK);
```



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Eccezioni e integrazione con type checking



Eccezioni

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
x :: s => ...
```



Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:



Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista



Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: :: e []



Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: `::` e `[]`
3. la definizione per casi di pop ha un caso solo per `::`
4. da cui il warning



Eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: `::` e `[]`
3. la definizione per casi di pop ha un caso solo per `::`
4. da cui il warning
5. il compilatore inserisce automaticamente una eccezione `Match` nei casi mancanti



Dichiarazione e generazione eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;          (* dichiara una nuova eccezione *)  
  
fun pop(x::s) = (x,s)  
| pop [] = raise EmptyStack;   (* come il throw di Java *)
```



Dichiarazione e generazione eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;          (* dichiara una nuova eccezione *)  
  
fun pop(x::s) = (x,s)  
| pop [] = raise EmptyStack;   (* come il throw di Java *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione "automatica" Match

```
- pop [];  
uncaught exception EmptyStack
```



Dichiarazione e generazione eccezioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;          (* dichiara una nuova eccezione *)  
  
fun pop(x::s) = (x,s)  
| pop [] = raise EmptyStack;   (* come il throw di Java *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione "automatica" Match

```
- pop [];  
uncaught exception EmptyStack
```

- Le eccezioni possono essere catturate e gestite con handle:

```
pop x  
handle EmptyStack =>  
  ( print "messaggio di errore specializzato";  
    raise EmptyStack );
```



Il costrutto handle

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Il costrutto handle

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo handle può gestire diverse eccezioni

```
<expression> handle  
  <exception 1> => ...  
  | <exception 2> => ...  
  | ...
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Il costrutto handle

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo handle può gestire diverse eccezioni

```
<expression> handle  
  <exception 1> => ...  
  | <exception 2> => ...  
  | ...
```

Ogni ordine è buono: perchè?

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il costrutto handle

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo handle può gestire diverse eccezioni

```
<expression> handle  
  <exception 1> => ...  
  | <exception 2> => ...  
  | ...
```

Ogni ordine è buono: perchè?

- Due modi di usarlo in ML funzionale puro:
 1. fare qualcosa come stampare un messaggio e *rilanciare l'eccezione*

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il costrutto handle

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo handle può gestire diverse eccezioni

```
<expression> handle  
  <exception 1> => ...  
  | <exception 2> => ...  
  | ...
```

Ogni ordine è buono: perchè?

- Due modi di usarlo in ML funzionale puro:

1. fare qualcosa come stampare un messaggio e *rilanciare l'eccezione*
2. “aggiustare” l’errore restituendo un valore *dello stesso tipo* dell’espressione che ha sollevato l’eccezione

Sono le uniche opzioni che passano il type checking senza errori

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il costrutto handle

■ Altro esempio (da non seguire acriticamente ...)

```
- fun pos x (y::z) =  
    if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Il costrutto handle

■ Altro esempio (da non seguire acriticamente ...)

```
- fun pos x (y::z) =  
    if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

■ Si può usare handle per modificare pos per restituire -1 quando non trova x nella lista:

```
- fun pos2 x y = (pos x y) handle Match => ~1;
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Il costrutto handle

■ Altro esempio (da non seguire acriticamente ...)

```
- fun pos x (y::z) =  
    if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

■ Si può usare handle per modificare pos per restituire -1 quando non trova x nella lista:

```
- fun pos2 x y = (pos x y) handle Match => ~1;
```

■ Esempio didattico un po' artificiale: si potrebbe obiettare che pos è realizzata male...

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Eccezioni con parametri

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni
- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Eccezioni con parametri

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni
- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

- Questa eccezione può essere lanciata in diversi modi...

```
raise SyntaxError "Identifier expected"
```

```
raise SyntaxError "Integer expected"
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Eccezioni con parametri

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni
- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

- Questa eccezione può essere lanciata in diversi modi...

```
raise SyntaxError "Identifier expected"
```

```
raise SyntaxError "Integer expected"
```

- ... e il parametro “letto” col pattern matching

```
... handle SyntaxError x => ... (* qui si può usare x *)
```



Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

Esempio: compilazione di espressioni



Elaborazione di simboli in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
 - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)



Elaborazione di simboli in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
 - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)
- Ne approfittiamo per dare un'idea parziale di alcune strutture dati interne al compilatore e dei procedimenti di generazione e ottimizzazione del codice



Elaborazione di simboli in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
 - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)
- Ne approfittiamo per dare un'idea parziale di alcune strutture dati interne al compilatore e dei procedimenti di generazione e ottimizzazione del codice
- L'esempio che segue realizza un compilatore per un linguaggio molto semplificato che supporta solo semplici espressioni su numeri interi. Il codice oggetto deve calcolare l'espressione data.



Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

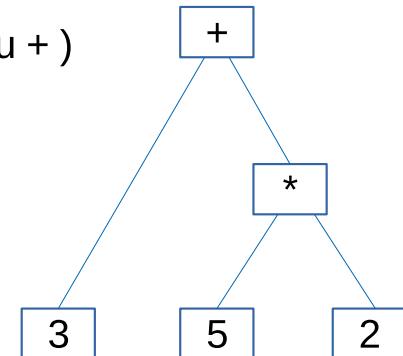
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)





Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

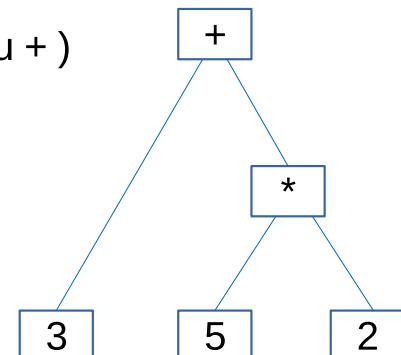
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

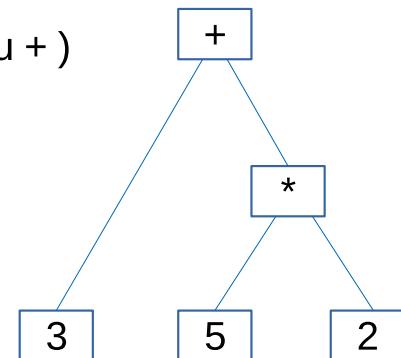
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Stack





Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

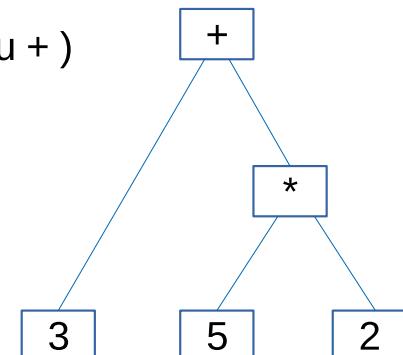
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Stack





Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

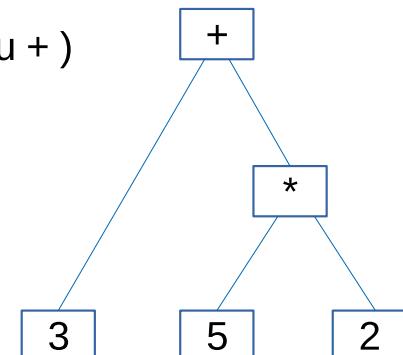
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

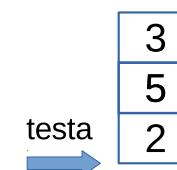
- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Stack





Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

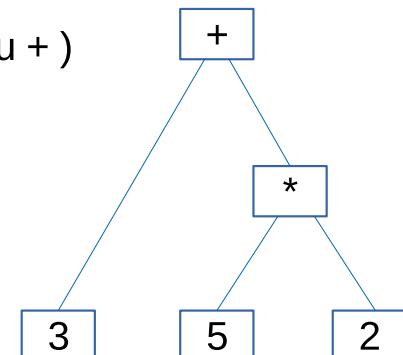
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

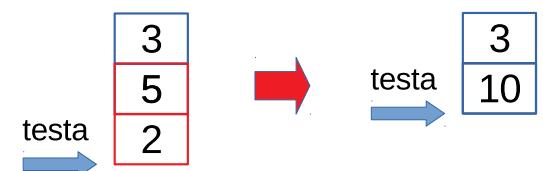
- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Stack





Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

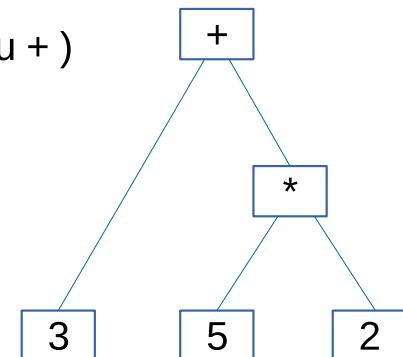
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

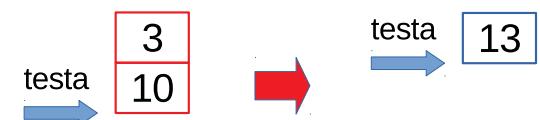
- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Stack





Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

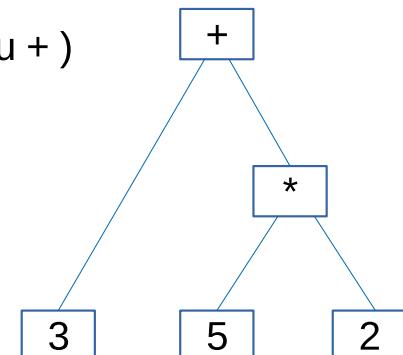
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione: $3 + 5 * 2$



Albero sintattico
(* ha precedenza su +)



Procedimento di calcolo:

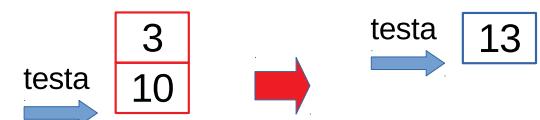
- salvare i risultati intermedi (come $5 * 2$) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
 - se sono su una foglia la metto sullo stack
 - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita: 3 5 2 * +



Stack



Il compilatore, a partire dall'albero sintattico, deve generare un codice che implementa questo procedimento



Definizione degli alberi sintattici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Introduciamo un costruttore per ogni operazione supportata dal linguaggio sorgente
- ogni costruttore corrisponde a un tipo di nodo dell'albero sintattico

```
datatype syntree = CO of int (* le costanti *)
                  | PLUS of syntree * syntree
                  | MINUS of syntree * syntree
                  | TIMES of syntree * syntree
                  | DIVIDE of syntree * syntree
                  | MODULUS of syntree * syntree
```



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int (* LOADC i c => Ri := c *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int (* LOADC i c => Ri := c *)
  | LOADI of int * int (* LOADI i j => Ri := mem(Rj) *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int (* LOADC i c => Ri := c *)
  | LOADI of int * int (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int (* STOREI i j => mem(Rj) := Ri *)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int     (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int              (* INCR i => Ri := Ri + 1 *)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int     (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int              (* INCR i => Ri := Ri + 1 *)
  | DECR of int              (* DECR i => Ri := Ri - 1 *)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int     (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int              (* INCR i => Ri := Ri + 1 *)
  | DECR of int              (* DECR i => Ri := Ri - 1 *)
  | SUM of int * int        (* SUM i j => Ri := Ri + Rj *)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



Definizione del linguaggio target

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int     (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int              (* INCR i => Ri := Ri + 1 *)
  | DECR of int              (* DECR i => Ri := Ri - 1 *)
  | SUM of int * int        (* SUM i j => Ri := Ri + Rj *)
  | SUB of int * int        (* SUB i j => Ri := Ri - Rj *)
  | MUL of int * int        (* MUL i j => Ri := Ri * Rj *)
  | DIV of int * int        (* DIV i j => Ri := Ri / Rj *)
  | MOD of int * int        (* MOD i j => Ri := Ri mod Rj *)
```



Definizione del linguaggio target

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int (* LOADC i c => Ri := c *)
  | LOADI of int * int (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int          (* INCR i => Ri := Ri + 1 *)
  | DECR of int          (* DECR i => Ri := Ri - 1 *)
  | SUM of int * int    (* SUM i j => Ri := Ri + Rj *)
  | SUB of int * int    (* SUB i j => Ri := Ri - Rj *)
  | MUL of int * int    (* MUL i j => Ri := Ri * Rj *)
  | DIV of int * int    (* DIV i j => Ri := Ri / Rj *)
  | MOD of int * int    (* MOD i j => Ri := Ri mod Rj *)
  | HALT
```

Obiettivo: terminare il programma con il risultato in cima allo stack.



Generazione del codice – funzione codegen

- Il codice oggetto utilizza tre registri:

- ◆ R1 come puntatore alla testa dello stack
- ◆ R2,R3 per calcolare le singole operazioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Generazione del codice – funzione codegen

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

- Il codice oggetto utilizza tre registri:
 - ◆ R1 come puntatore alla testa dello stack
 - ◆ R2,R3 per calcolare le singole operazioni
- La traduzione vera e propria è effettuata da una funzione ausiliaria `translate` che prende in input:
 - ◆ un albero sintattico `tree`
 - ◆ una *continuazione*, ovvero il codice da eseguire *dopo* avere eseguito le operazioni contenute in `tree`



Generazione del codice – funzione codegen

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

- Il codice oggetto utilizza tre registri:
 - ◆ R1 come puntatore alla testa dello stack
 - ◆ R2,R3 per calcolare le singole operazioni
- La traduzione vera e propria è effettuata da una funzione ausiliaria `translate` che prende in input:
 - ◆ un albero sintattico `tree`
 - ◆ una *continuazione*, ovvero il codice da eseguire *dopo* avere eseguito le operazioni contenute in `tree`
- Pertanto la prima chiamata a `translate` gli passerà
 - ◆ l'albero sintattico dell'intera espressione da compilare
 - ◆ la lista di istruzioni [HALT]



Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =
```



Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =
let
  (* definizione della funzione translate *)
  (* R1: stack pointer; R2: accumulator *)
```



Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =
let
  (* definizione della funzione translate *)
  (* R1: stack pointer; R2: accumulator *)
  fun translate (co x) cont =
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
```



Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =
let
  (* definizione della funzione translate *)
  (* R1: stack pointer; R2: accumulator *)
  fun translate (co x) cont =
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
  | translate (plus(t1,t2)) cont =
    translate t1 (
      translate t2 (
        LOADI(2,1)::DECR(1)::LOADI(3,1)::SUM(2,3)::STOREI(2,1)::cont))
```



Generazione del codice

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
fun codegen tree =
let
  (* definizione della funzione translate *)
  (* R1: stack pointer; R2: accumulator *)
  fun translate (co x) cont =
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
  | translate (plus(t1,t2)) cont =
    translate t1 (
      translate t2 (
        LOADI(2,1)::DECR(1)::LOADI(3,1)::SUM(2,3)::STOREI(2,1)::cont))
  | translate (times(t1,t2)) cont = simile ma con MUL al posto di SUM
  | translate (minus(t1,t2)) cont = simile ma con SUB(3,2) al posto di SUM(2,3)
  | translate (divide(t1,t2)) cont = simile ma con DIV al posto di SUB
  | translate (modulus(t1,t2)) cont = simile ma con MOD al posto di SUB
```



Generazione del codice

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
fun codegen tree =
let
  (* definizione della funzione translate *)
  (* R1: stack pointer; R2: accumulator *)
  fun translate (co x) cont =
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
  | translate (plus(t1,t2)) cont =
    translate t1 (
      translate t2 (
        LOADI(2,1)::DECR(1)::LOADI(3,1)::SUM(2,3)::STOREI(2,1)::cont))
  | translate (times(t1,t2)) cont = simile ma con MUL al posto di SUM
  | translate (minus(t1,t2)) cont = simile ma con SUB(3,2) al posto di SUM(2,3)
  | translate (divide(t1,t2)) cont = simile ma con DIV al posto di SUB
  | translate (modulus(t1,t2)) cont = simile ma con MOD al posto di SUB
in
  translate tree [HALT]
end
```



Ottimizzazione del codice

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La generazione meccanica introduce operazioni inutili.

```
LOADC 2 3      (* codice generato per 3+5*2 *)
```

```
INCR 1
```

```
STOREI 2 1
```

```
—
```

```
LOADC 2 5
```

```
INCR 1
```

```
STOREI 2 1
```

```
—
```

```
LOADC 2 2
```

```
INCR 1
```

```
STOREI 2 1
```

```
—
```

```
LOADI 2 1
```

```
DECR 1
```

```
LOADI 3 1
```

```
MUL 2 3
```

```
STOREI 2 1
```

```
—
```

```
LOADI 2 1
```

```
DECR 1
```

```
LOADI 3 1
```

```
SUM 2 3
```

```
STOREI 2 1
```

```
—
```

```
HALT
```



Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione `optimize` elimina le più comuni operazioni ridondanti
- Itera una funzione ausiliaria `opt1` che esegue un singolo passo di ottimizzazione
- Questo può attivare ulteriori semplificazioni, quindi `optimize` itera `opt1` finché il codice non può essere ulteriormente ridotto



Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

local

(* definizione singolo passo di ottimizzazione *)



Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
```



Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
      if x=y
        then cont'
        else INCR(x)::DECR(y)::cont'
    end
```



Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
      if x=y
        then cont'
        else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
      if x=x1 andalso y=y1
        then STOREI(x,y)::cont'
        else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
```



Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
      if x=y
        then cont'
        else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
      if x=x1 andalso y=y1
        then STOREI(x,y)::cont'
        else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
  | (* altri casi qui... *)
  | opt1 (c :: cont) = c :: (opt1 cont)
```



Ottimizzazione del codice – funzione optimize

Paradigma
funzionale

ML

Dichiarazioni e
scoping in ML

Tipi strutturati in
ML

Patterns e matching

Liste

Currying

Funzioni di ordine
superiore

Polimorfismo
parametrico

Encapsulation e
interfacce

Eccezioni e
integrazione con
type checking

Esempio: un
semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
      if x=y
        then cont'
        else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
      if x=x1 andalso y=y1
        then STOREI(x,y)::cont'
        else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
  | (* altri casi qui... *)
  | opt1 (c :: cont) = c :: (opt1 cont)
in
  fun optimize code =
    let val code' = opt1 code in      (* fa 1 passo di ottimizz.*)
      if length(code') = length(code) (* se nessun progresso *)
        then code'                  (* termina *)
        else optimize code'          (* altrimenti riprova *)
    end
  end
```



Efficacia dell'ottimizzazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ecco il risultato per la solita espressione $3+5*2$. A sinistra il codice non ottimizzato, a destra quello ottimizzato

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
INCR 1
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
LOADI 3 1
MUL 2 3
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

Guadagno: 30% di istruzioni in meno



Combinare le fasi con composizione di funzioni

- L'operatore \circ denota la composizione di funzioni

- ◆ $(f \circ g)(x) = f(g(x))$

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Combinare le fasi con composizione di funzioni

- L'operatore `o` denota la composizione di funzioni

- ◆ $(f \circ g)(x) = f(g(x))$

- Con la composizione è facile definire l'intero processo di compilazione assemblando le diverse fasi

```
- val compile = optimize o codegen o parse;  
  
val it = fn : string -> instruction list
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Conclusioni sull'esempio di compilazione

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
 - ◆ in Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato



Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
 - ◆ in Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato
- Inoltre la type inference ci permette di omettere il tipo degli identificatori, producendo un codice più snello
 - ◆ come fosse uno scripting language debolmente tipato
 - ◆ ma senza sacrificare il controllo di tipi forte



Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
 - ◆ in Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato
- Inoltre la type inference ci permette di omettere il tipo degli identificatori, producendo un codice più snello
 - ◆ come fosse uno scripting language debolmente tipato
 - ◆ ma senza sacrificare il controllo di tipi forte
- Per queste ragioni linguaggi come ML vengono utilizzati per la prototipizzazione rapida di compilatori e interpreti

Linguaggi di Programmazione I – Lezione 18

Proff. Piero Bonatti e Marco Faella

<mailto://pab@unina.it>

<mailto://marfaella@gmail.com>

7 giugno 2022



Panoramica dell'argomento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

Paradigma logico



L'essenza del paradigma logico

programmi = insiemi di assiomi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

programmi = insiemi di assiomi

computazioni = dimostrazioni costruttive di una formula logica
data – detta *query* – mediante gli assiomi del programma



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

programmi = insiemi di assiomi

computazioni = dimostrazioni costruttive di una formula logica data – detta *query* – mediante gli assiomi del programma

Esempio

■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

programmi = insiemi di assiomi

computazioni = dimostrazioni costruttive di una formula logica data – detta *query* – mediante gli assiomi del programma

Esempio

■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

■ Query 1

- ◆ Socrate è un uomo?



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

programmi = insiemi di assiomi

computazioni = dimostrazioni costruttive di una formula logica data – detta *query* – mediante gli assiomi del programma

Esempio

■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

■ Query 1

- ◆ Socrate è un uomo?
- ◆ risposta: *true*



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

programmi = insiemi di assiomi

computazioni = dimostrazioni costruttive di una formula logica data – detta *query* – mediante gli assiomi del programma

Esempio

■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

■ Query 1

- ◆ Socrate è un uomo?
- ◆ risposta: *true*

■ Query 2

- ◆ Socrate è mortale?



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

programmi = insiemi di assiomi

computazioni = dimostrazioni costruttive di una formula logica data – detta *query* – mediante gli assiomi del programma

Esempio

■ Programma:

- ◆ Socrate è un uomo. (assioma 1)
- ◆ Tutti gli uomini sono mortali. (assioma 2)

■ Query 1

- ◆ Socrate è un uomo?
- ◆ risposta: *true*

■ Query 2

- ◆ Socrate è mortale?
- ◆ risposta: *true*



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Esempio in Prolog

■ Programma:

- ◆ `umano(socrate).` (fatto)
- ◆ `mortale(X) :- umano(X).` (regola)



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Esempio in Prolog

■ Programma:

- ◆ umano(socrate). (fatto)
- ◆ mortale(X) :- umano(X). (regola)

■ Query 1

- ◆ umano(socrate).
- ◆ risposta: *true*



L'essenza del paradigma logico

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Esempio in Prolog

■ Programma:

- ◆ umano(socrate). (fatto)
- ◆ mortale(X) :- umano(X). (regola)

■ Query 1

- ◆ umano(socrate).
- ◆ risposta: *true*

■ Query 2

- ◆ mortale(socrate).
- ◆ risposta: *true*



Prolog

[Paradigma logico](#)

Prolog

[Implementazione](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- Illustreremo assiomi e query mediante il linguaggio di programmazione logica *Prolog*
- Basandoci sul libro *The Art of Prolog* di Sterling e Shapiro, seconda edizione
 - ◆ reperibile in biblioteca e on-line



Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

Implementazione

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

- L'implementazione è mista:



Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

Implementazione

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

■ L'implementazione è mista:

- ◆ I programmi Prolog vengono compilati in un bytecode
- ◆ che viene interpretato da una macchina virtuale
- ◆ la *Warren abstract machine* (WAM)



Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

Implementazione

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

- L'implementazione è mista:
 - ◆ I programmi Prolog vengono compilati in un bytecode
 - ◆ che viene interpretato da una macchina virtuale
 - ◆ la *Warren abstract machine* (WAM)
 - ◆ si possono anche compilare i programmi in codice stand-alone, direttamente eseguibile
- L'interazione con Prolog è analoga a quella con ML
 - ◆ si inviano query all'interprete e si ottengono le relative risposte



Implementazione e Interazione in Prolog

[Paradigma logico](#)

[Prolog](#)

Implementazione

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Analoghe a quelle di ML:

- L'implementazione è mista:
 - ◆ I programmi Prolog vengono compilati in un bytecode
 - ◆ che viene interpretato da una macchina virtuale
 - ◆ la *Warren abstract machine* (WAM)
 - ◆ si possono anche compilare i programmi in codice stand-alone, direttamente eseguibile
- L'interazione con Prolog è analoga a quella con ML
 - ◆ si inviano query all'interprete e si ottengono le relative risposte
 - ◆ oppure, se il programma è stand alone, si interagisce mediante la sua UI (user interface)



Sistema consigliato per il corso

■ SWI Prolog (free)

- ◆ implementa il Prolog standard
- ◆ supporta sia interpretazione che compilazione stand-alone

[Paradigma logico](#)

[Prolog](#)

[**Implementazione**](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Sistema consigliato per il corso

Paradigma logico

Prolog

Implementazione

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

■ SWI Prolog (free)

- ◆ implementa il Prolog standard
- ◆ supporta sia interpretazione che compilazione stand-alone
- ◆ invocazione da command line:

```
$ swipl  
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
?-
```

(?- è il prompt dell'interprete)



Sistema consigliato per il corso

Paradigma logico

Prolog

Implementazione

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

■ SWI Prolog (free)

- ◆ implementa il Prolog standard
- ◆ supporta sia interpretazione che compilazione stand-alone
- ◆ invocazione da command line:

```
$ swipl  
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)  
?-
```

(?- è il prompt dell'interprete)

■ per caricare un proprio programma mioprog.pl

| | |
|-------------------------------------|--|
| ?- 'mioprog.pl' . | oppure |
| ?- consult('mioprog.pl') . | quando si carica la prima volta; |
| ?- reconsult('mioprog.pl') . | quando si ricarica dopo una correzione |



Costrutti base

■ Tre tipi di statement:

1. fatti (facts)
2. regole (rules)
3. queries (detti anche goals)

[Paradigma logico](#)

[Prolog](#)

[**Costrutti Prolog**](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Costrutti base

■ Tre tipi di statement:

1. fatti (facts)
2. regole (rules)
3. queries (detti anche goals)

■ Una sola struttura dati:

1. termini logici (logical terms)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Fatti

- Asseriscono una relazione tra oggetti

```
father(abraham, isaac). (Abramo è il padre di Isacco)
```

father, oltre che *relazione*, è chiamato anche *predicato*

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Fatti

- Asseriscono una relazione tra oggetti

```
father(abraham, isaac). (Abramo è il padre di Isacco)
```

father, oltre che *relazione*, è chiamato anche *predicato*

- Altro esempio: le tabelline

```
per(2,1,2). (due per uno due)  
per(2,2,4). (due per due quattro)  
per(2,3,6). (due per tre sei)  
...
```

- I nomi dei predicati devono iniziare per lettera minuscola

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

[Queries](#)

[Variabili logiche](#)

[I termini](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Fatti

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- Asseriscono una relazione tra oggetti

```
father(abraham, isaac). (Abramo è il padre di Isacco)
```

father, oltre che *relazione*, è chiamato anche *predicato*

- Altro esempio: le tabelline

```
per(2,1,2). (due per uno due)  
per(2,2,4). (due per due quattro)  
per(2,3,6). (due per tre sei)  
...
```

- I nomi dei predicati devono iniziare per lettera minuscola
- Gli argomenti abraham e isaac iniziano con lettera minuscola perchè sono *costanti*, come i numeri
 - ◆ introdurremo le *variabili* più avanti



Fatti

- Con i fatti possiamo definire un *database*
 - ◆ l'esempio più semplice di *programma logico*

| | |
|--------------------------|-----------------|
| father(terach, abraham). | male(terach). |
| father(terach, nachor). | male(abraham). |
| father(terach, haran). | male(nachor). |
| father(abraham, isaac). | male(haran). |
| father(haran, lot). | male(isaac). |
| father(haran, milcah). | male(lot). |
| father(haran, yiscah). | |
| | female(sarah). |
| mother(sarah, isaac). | female(milcah). |
| | female(yiscah). |

Program 1.1 A biblical family database

Ogni predicato corrisponde a una *tabella relazionale*



Queries

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).  
true.
```

```
?- father(isaac,abraham).  
false.
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

Queries

[Variabili logiche](#)

[I termini](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

Unificazione

[Conjunctive queries](#)

[Le Regole](#)

Ragionamento

Liste

[Applicazioni](#)

Programmazione
nondeterministica

Unicità di Prolog

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).  
true.
```

```
?- father(isaac,abraham).  
false.
```

- Le query hanno la stessa forma dei fatti. Sono entrambi dei cosiddetti *atomi logici*.
 - ◆ nei programmi sono asserzioni (fatti)
 - ◆ nelle query sono domande



Queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

Queries

[Variabili logiche](#)

[I termini](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).  
true.
```

```
?- father(isaac,abraham).  
false.
```

- Le query hanno la stessa forma dei fatti. Sono entrambi dei cosiddetti *atomi logici*.
 - ◆ nei programmi sono asserzioni (fatti)
 - ◆ nelle query sono domande
- Nell'esempio qui sopra, l'interprete trova il primo atomo nel programma e non trova il secondo (perciò le risposte diverse)



Queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

Queries

[Variabili logiche](#)

[I termini](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- I programmi logici sono fatti per rispondere a *queries*
- Se il programma caricato è quello nella slide precedente allora:

```
?- father(abraham,isaac).  
true.
```

```
?- father(isaac,abraham).  
false.
```

- Le query hanno la stessa forma dei fatti. Sono entrambi dei cosiddetti *atomi logici*.
 - ◆ nei programmi sono asserzioni (fatti)
 - ◆ nelle query sono domande
- Nell'esempio qui sopra, l'interprete trova il primo atomo nel programma e non trova il secondo (perciò le risposte diverse)
- Nota: nel libro le query sono indicate con un punto interrogativo dopo l'atomo, come in `father(abraham,isaac)?`



Variabili logiche nelle query

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
 - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

```
?- father(abraham,X).      /* esiste X tale che father(abraham,X) ? */  
X=isaac.
```



Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
 - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

```
?- father(abraham,X).      /* esiste X tale che father(abraham,X) ? */  
X=isaac.
```

```
?- father(terach,X).      /* esiste X tale che father(terach,X) ? */  
X=abraham
```



Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
 - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

```
?- father(abraham,X).      /* esiste X tale che father(abraham,X) ? */  
X=isaac.
```

```
?- father(terach,X).      /* esiste X tale che father(terach,X) ? */  
X=abraham;  
X=nachor
```



Variabili logiche nelle query

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- È utile chiedere cose come: *quali sono i figli di abraham?*
- Si può fare con le *variabili logiche*
 - ◆ che si riconoscono perchè iniziano con una *lettera maiuscola*

```
?- father(abraham,X).      /* esiste X tale che father(abraham,X) ? */  
X=isaac.  
  
?- father(terach,X).        /* esiste X tale che father(terach,X) ? */  
X=abraham;  
X=nachor;  
X=haran.
```

- La macchina virtuale cerca i valori che – sostituiti a X – rendono la query uguale a uno dei fatti nel programma
 - ◆ NB: questo vale per i programmi di soli fatti...



Variabili logiche in generale

- Differenza tra le variabili logiche e le variabili degli altri paradigmi
 - ◆ Le variabili logiche rappresentano *oggetti qualsiasi*, non specificati
 - ◆ Le variabili dei linguaggi imperativi sono locazioni di memoria
 - ◆ Gli identificatori dei linguaggi funzionali denotano valori immutabili

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Variabili logiche in generale

Paradigma logico

Prolog

Costrutti Prolog

Fatti

Queries

Variabili logiche

I termini

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

■ Differenza tra le variabili logiche e le variabili degli altri paradigmi

- ◆ Le variabili logiche rappresentano *oggetti qualsiasi*, non specificati
- ◆ Le variabili dei linguaggi imperativi sono locazioni di memoria
- ◆ Gli identificatori dei linguaggi funzionali denotano valori immutabili

■ Si possono usare anche nei fatti. Un credente potrebbe scrivere in un programma:

```
creato_da(X,dio). /* ogni cosa è creata da Dio */
```

■ Differenza tra le variabili nei fatti e nelle query:

- ◆ nelle query: *esistenzialmente quantificate*
 - *esiste* un X tale che ... ?
- ◆ nei fatti: *universalmente quantificate*
 - *per ogni* X vale che ...



I termini

- I termini sono l'unica struttura dati in Prolog
- Si costruiscono con costanti, variabili (logiche) e *funtori*
 - ◆ ruolo simile ai costruttori di ML
 - ◆ caratterizzati da nome e *arietà* (il numero di argomenti)

```
<term> ::= <constant> | <variable>
          | <functor name> '(' [<term> [ ',' <term>]*] ')'
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



I termini

- I termini sono l'unica struttura dati in Prolog
- Si costruiscono con costanti, variabili (logiche) e *funtori*
 - ◆ ruolo simile ai costruttori di ML
 - ◆ caratterizzati da nome e *arietà* (il numero di argomenti)

```
<term> ::= <constant> | <variable>
          | <functor name> '(' [<term> [ ',' <term>]*] ')'
```

Esempi

```
successor(Int)
date(25,april,2020)
color(rgb,0,0,1)
```

- Non ci sono dichiarazioni di tipo
- Costanti simboliche e funtori si usano senza dichiararli prima.



I termini

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october,2000)).  
hasColor(object1 , color(rgb,1,0,0)).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



I termini

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october,2000)).  
hasColor(object1 , color(rgb,1,0,0)).
```

- Quindi la sintassi generale dei fatti è

```
<fact> ::= <atomic formula> '.'  
  
<atomic formula> ::=  
    <predicate> '(' [<term> [ ',' <term> ] * ] ')'.
```

(notare il punto dopo il fatto)



I termini

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october,2000)).  
hasColor(object1 , color(rgb,1,0,0)).
```

- Quindi la sintassi generale dei fatti è

```
<fact> ::= <atomic formula> '.'  
  
<atomic formula> ::=  
    <predicate> '(' [<term> [ ',' <term>]*] ')'.
```

(notare il punto dopo il fatto)

- Esempi di query a un programma coi fatti qui sopra

```
?- born(X, date(Y,october,2000)).  
X=john ,  
Y=10 .
```



I termini

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Gli argomenti di un predicato possono essere termini qualsiasi

```
born(john , date(10,october,2000)).  
hasColor(object1 , color(rgb,1,0,0)).
```

- Quindi la sintassi generale dei fatti è

```
<fact> ::= <atomic formula> '.'  
  
<atomic formula> ::=  
    <predicate> '(' [<term> [ ',' <term>]*] ')'
```

(notare il punto dopo il fatto)

- Esempi di query a un programma coi fatti qui sopra

```
?- born(X, date(Y,october,2000)).
```

```
X=john ,  
Y=10 .
```

```
?- hasColor(object1 , Y).
```

```
Y=color(rgb,1,0,0).
```



I termini

- La stessa variabile X può comparire più volte nello stesso fatto o nella stessa query
- Significa che i termini nelle posizioni dove si trova X devono essere uguali tra loro

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



I termini

- La stessa variabile X può comparire più volte nello stesso fatto o nella stessa query
- Significa che i termini nelle posizioni dove si trova X devono essere uguali tra loro

```
/* fatti */  
div(X,X,1).  
sum(X,0,X).  
sum(0,X,X).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



I termini

- La stessa variabile X può comparire più volte nello stesso fatto o nella stessa query
- Significa che i termini nelle posizioni dove si trova X devono essere uguali tra loro

```
/* fatti */  
div(X,X,1).  
sum(X,0,X).  
sum(0,X,X).  
  
/* query */  
?- hasColor(object1, color(rgb,Y,Y,Y)).  
    false.                                /* nell'unico fatto su object1 c'è 1,0,0 */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



I termini

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Fatti

Queries

Variabili logiche

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- La stessa variabile X può comparire più volte nello stesso fatto o nella stessa query
- Significa che i termini nelle posizioni dove si trova X devono essere uguali tra loro

```
/* fatti */
div(X,X,1).
sum(X,0,X).
sum(0,X,X).

/* query */
?- hasColor(object1, color(rgb,Y,Y,Y)).
   false.                      /* nell'unico fatto su object1 c'è 1,0,0 */
```

- Differenza tra termini di Prolog e pattern di ML
 - ◆ in ML non posso usare la stessa variabile più volte nello stesso pattern
 - ◆ perchè servono solo a estrarre informazioni da una struttura, non a controllare se elementi diversi sono uguali



Ground e nonground

- Un termine è *ground* se non contiene variabili
- Altrimenti è *nonground*

```
foo(a,b)    /* ground      */  
bar(X)      /* nonground  */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

[Queries](#)

[Variabili logiche](#)

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Ground e nonground

- Un termine è *ground* se non contiene variabili
- Altrimenti è *nonground*

```
foo(a,b)      /* ground      */  
bar(X)        /* nonground */
```

- Gli stessi aggettivi e gli stessi criteri si applicano ai fatti, alle query e anche alle regole (che vedremo più avanti)

```
father(abraham,isaac).    /* ground      */  
sum(X,0,X).                /* nonground */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Fatti](#)

[Queries](#)

[Variabili logiche](#)

I termini

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Unificazione



Come si risponde alle query

- Per spiegarlo dobbiamo specificare come avviene il matching tra query e fatti e come si costruiscono le risposte
- Dovremo introdurre due nozioni:
 - ◆ sostituzione
 - ◆ unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Sostituzioni

- Una sostituzione è un insieme finito di coppie

$$\theta = \{X_1 = t_1, \dots X_n = t_n\}$$

dove

- ◆ Le X_i sono variabili e i t_i termini.
- ◆ Le X_i sono tutte diverse tra loro.
- ◆ Nessuna delle X_i compare dentro i t_i .

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Sostituzioni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Una sostituzione è un insieme finito di coppie

$$\theta = \{X_1 = t_1, \dots X_n = t_n\}$$

dove

- ◆ Le X_i sono variabili e i t_i termini.
 - ◆ Le X_i sono tutte diverse tra loro.
 - ◆ Nessuna delle X_i compare dentro i t_i .
- L'applicazione di θ a una espressione E (che potrebbe essere un termine, un fatto, una query o una regola)
 - ◆ si denota con $E\theta$
 - ◆ sostituisce le occorrenze delle variabili X_i in E con i rispettivi termini t_i



Sostituzioni

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Una sostituzione è un insieme finito di coppie

$$\theta = \{X_1 = t_1, \dots X_n = t_n\}$$

dove

- ◆ Le X_i sono variabili e i t_i termini.
- ◆ Le X_i sono tutte diverse tra loro.
- ◆ Nessuna delle X_i compare dentro i t_i .

- L'applicazione di θ a una espressione E (che potrebbe essere un termine, un fatto, una query o una regola)
 - ◆ si denota con $E\theta$
 - ◆ sostituisce le occorrenze delle variabili X_i in E con i rispettivi termini t_i
- Esempio: se $\theta = \{x = isaac\}$ e $E = \text{father}(abraham, x)$ allora

$$E\theta = \text{father}(abraham, isaac)$$



Istanze

- L'applicazione di una sostituzione θ a E crea un “caso particolare” di E ,
 - ◆ dove le variabili di E (che indicano oggetti non specificati)
 - ◆ vengono sostituite con valori specifici (termini ground)
 - ◆ o parzialmente specificati (termini nonground)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Istanze

- L'applicazione di una sostituzione θ a E crea un “caso particolare” di E ,
 - ◆ dove le variabili di E (che indicano oggetti non specificati)
 - ◆ vengono sostituite con valori specifici (termini ground)
 - ◆ o parzialmente specificati (termini nonground)
- Esempio: $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$ e $E = \text{hasColor(o1, X)}$.

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

Istanze

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Istanze

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

Istanze

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- L'applicazione di una sostituzione θ a E crea un “caso particolare” di E ,
 - ◆ dove le variabili di E (che indicano oggetti non specificati)
 - ◆ vengono sostituite con valori specifici (termini ground)
 - ◆ o parzialmente specificati (termini nonground)
- Esempio: $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$ e $E = \text{hasColor(o1, X)}$.
 - ◆ $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$



Istanze

- L'applicazione di una sostituzione θ a E crea un “caso particolare” di E ,
 - ◆ dove le variabili di E (che indicano oggetti non specificati)
 - ◆ vengono sostituite con valori specifici (termini ground)
 - ◆ o parzialmente specificati (termini nonground)
- Esempio: $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$ e $E = \text{hasColor(o1, X)}$.
 - ◆ $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$
 - ◆ E dice che o1 ha un colore non specificato

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

Istanze

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Istanze

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

Istanze

[L'unificazione](#)

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- L'applicazione di una sostituzione θ a E crea un “caso particolare” di E ,
 - ◆ dove le variabili di E (che indicano oggetti non specificati)
 - ◆ vengono sostituite con valori specifici (termini ground)
 - ◆ o parzialmente specificati (termini nonground)
- Esempio: $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$ e $E = \text{hasColor(o1, X)}$.
 - ◆ $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$
 - ◆ E dice che o1 ha un colore non specificato
 - ◆ $E\theta$ lo specifica parzialmente: è in formato rgb e tutti i 3 valori sono uguali



Istanze

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- L'applicazione di una sostituzione θ a E crea un “caso particolare” di E ,
 - ◆ dove le variabili di E (che indicano oggetti non specificati)
 - ◆ vengono sostituite con valori specifici (termini ground)
 - ◆ o parzialmente specificati (termini nonground)
- Esempio: $\theta = \{X = \text{color(rgb, Y, Y, Y)}\}$ e $E = \text{hasColor(o1, X)}$.
 - ◆ $E\theta = \text{hasColor(o1, color(rgb, Y, Y, Y))}$
 - ◆ E dice che o1 ha un colore non specificato
 - ◆ $E\theta$ lo specifica parzialmente: è in formato rgb e tutti i 3 valori sono uguali
- Definizione: E_1 è un'istanza di E_2 se esiste θ tale che $E_1 = E_2\theta$
 - ◆ cioè se E_1 è un “caso particolare” di E_2 dove alcune variabili di E_2 sono istanziate, cioè legate a un valore



L'unificazione

- L'algoritmo di unificazione è quello che effettua il matching
- Prende due espressioni E_1 ed E_2 e – se possibile – restituisce una sostituzione θ tale che

$$E_1\theta = E_2\theta$$

detta *unificatore* (unifier)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- L'algoritmo di unificazione è quello che effettua il matching
- Prende due espressioni E_1 ed E_2 e – se possibile – restituisce una sostituzione θ tale che

$$E_1\theta = E_2\theta$$

detta *unificatore* (unifier)

- L'unificatore costruito dall'algoritmo viene detto *most general unifier* (mgu) perchè
 - ◆ non sostituisce una variabile con un termine se non è necessario, cioè
 - ◆ vincola il meno possibile il risultato $E_1\theta$, lasciando le variabili libere quando può



L'unificazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- L'algoritmo di unificazione è quello che effettua il matching
- Prende due espressioni E_1 ed E_2 e – se possibile – restituisce una sostituzione θ tale che

$$E_1\theta = E_2\theta$$

detta *unificatore* (unifier)

- L'unificatore costruito dall'algoritmo viene detto *most general unifier* (mgu) perchè
 - ◆ non sostituisce una variabile con un termine se non è necessario, cioè
 - ◆ vincola il meno possibile il risultato $E_1\theta$, lasciando le variabili libere quando può
 - ◆ tecnicamente, ogni altro unificatore θ' porta a una *istanza* (cioè caso particolare) di $E_1\theta$, cioè
 - ◆ esiste una sostituzione σ tale che $(E_1\theta)\sigma = E_1\theta'$



L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Esempi

- ◆ Le seguenti sostituzioni sono tutte unificatori delle espressioni $E_1 = \text{sum}(A, B, C)$ ed $E_2 = \text{sum}(X, 0, X)$:
 - $\theta_0 = \{A = 0, B = 0, C = 0, X = 0\}$
 - $\theta_1 = \{A = 1, B = 0, C = 1, X = 1\}$
 - etc.
 - $\sigma = \{A = X, B = 0, C = X\}$
- ◆ ...ma una sola è la *most general* (mgu)
- ◆ $\text{mgu}(\text{sum}(A, B, C), \text{sum}(X, 0, X)) = \{A = X, B = 0, C = X\}$



L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

Programmazione
nondeterministica

[Unicità di Prolog](#)

Esempi

- ◆ Le seguenti sostituzioni sono tutte unificatori delle espressioni $E_1 = \text{sum}(A, B, C)$ ed $E_2 = \text{sum}(X, 0, X)$:
 - $\theta_0 = \{A = 0, B = 0, C = 0, X = 0\}$
 - $\theta_1 = \{A = 1, B = 0, C = 1, X = 1\}$
 - etc.
 - $\sigma = \{A = X, B = 0, C = X\}$
- ◆ ...ma una sola è la *most general* (mgu)
 - ◆ $\text{mgu}(\text{sum}(A, B, C), \text{sum}(X, 0, X)) = \{A = X, B = 0, C = X\}$
 - ◆ $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, 0, Y)) = \{X = 0, Y = 0\}$



L'unificazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

[Applicazioni](#)

Programmazione
nondeterministica

[Unicità di Prolog](#)

Esempi

- ◆ Le seguenti sostituzioni sono tutte unificatori delle espressioni $E_1 = \text{sum}(A, B, C)$ ed $E_2 = \text{sum}(X, 0, X)$:
 - $\theta_0 = \{A = 0, B = 0, C = 0, X = 0\}$
 - $\theta_1 = \{A = 1, B = 0, C = 1, X = 1\}$
 - etc.
 - $\sigma = \{A = X, B = 0, C = X\}$
- ◆ ...ma una sola è la *most general* (mgu)
 - ◆ $\text{mgu}(\text{sum}(A, B, C), \text{sum}(X, 0, X)) = \{A = X, B = 0, C = X\}$
 - ◆ $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, 0, Y)) = \{X = 0, Y = 0\}$
 - ◆ $\text{mgu}(\text{sum}(0, X, 0), \text{sum}(Y, Z, 1))$ non esiste a causa del terzo argomento (non si può rendere $0=1$)



Esercizi sull'unificazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

1. Quali delle seguenti sostituzioni sono unificatori per le espressioni
 $E_1 = \text{hasColor}(X, \text{color}(\text{rgb}, \text{Red}, 0, \text{Blue}))$ e
 $E_2 = \text{hasColor}(\text{box}, \text{color}(\text{Format}, 0, Y, Y))$:
 - (a) $\{\text{Red} = 0\}$
 - (b) $\{\text{Red} = Z, Z = 0, \text{Blue} = Y\}$
 - (c) $\{X = \text{box}, \text{Red} = 0, \text{Blue} = Y\}$
 - (d) $\{X = \text{box}, \text{Red} = 0, \text{Blue} = 0, Y = 0\}$
 - (e) $\{X = \text{box}, \text{Red} = 0, \text{Blue} = 1, Y = 1, \text{Format} = \text{rgb}\}$
 - (f) $\{X = \text{box}, \text{Red} = 0, \text{Blue} = 0, Y = 0, \text{Format} = \text{rgb}\}$
2. Elencare almeno 3 unificatori, tra cui l'mgu, delle seguenti espressioni:
 $E_1 = \text{has_type}(a, \text{record}(\text{array}(X), Y, \text{pointer}(Z)))$,
 $E_2 = \text{has_type}(V, \text{record}(T, \text{float}, \text{pointer}(\text{void})))$



Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto $p(u_1, \dots, u_m)$ con lo stesso funtore (cioè $p = q$ e $m = n$). Se se ne trova uno:

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Sostituzioni](#)

[Istanze](#)

L'unificazione

[Search tree](#)

[Conjunctive queries](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto $p(u_1, \dots, u_m)$ con lo stesso funtore (cioè $p = q$ e $m = n$). Se se ne trova uno:
2. si calcola $\theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
3. se θ esiste, allora:

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto $p(u_1, \dots, u_m)$ con lo stesso funtore (cioè $p = q$ e $m = n$). Se se ne trova uno:
 - ◆ si calcola $\theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
 - 3. se θ esiste, allora:
 - ◆ si restituisce θ come risposta (se è vuota allora Prolog dice *true*)
 - ◆ poi se l'utente non vuole altre soluzioni si termina
4. altrimenti si cerca il prossimo fatto con lo stesso funtore. Se esiste si ripete da 2, altrimenti si termina

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Costruzione delle riposte da soli fatti

- Nei programmi visti sinora (che consistono di soli fatti) le risposte a una query

$$q(t_1, \dots, t_n)$$

vengono costruite così

1. si cerca nel programma il primo fatto $p(u_1, \dots, u_m)$ con lo stesso funtore (cioè $p = q$ e $m = n$). Se se ne trova uno:
 - ◆ si calcola $\theta = \text{mgu}(q(t_1, \dots, t_n), p(u_1, \dots, u_m))$
 - 3. se θ esiste, allora:
 - ◆ si restituisce θ come risposta (se è vuota allora Prolog dice *true*)
 - ◆ poi se l'utente non vuole altre soluzioni si termina
 4. altrimenti si cerca il prossimo fatto con lo stesso funtore. Se esiste si ripete da 2, altrimenti si termina
- Prolog dice *false* quando nessun fatto unifica con la query



Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

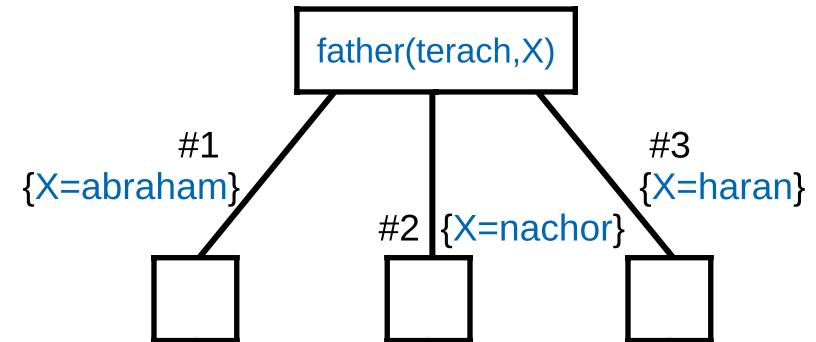
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Un *search tree* per la query `father(terach,X)`.

```
/* programma */  
/*1*/ father(terach, abraham).  
/*2*/ father(terach, nachor).  
/*3*/ father(terach, haran).  
/*4*/ father(abraham, isaac).  
    ...
```





Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

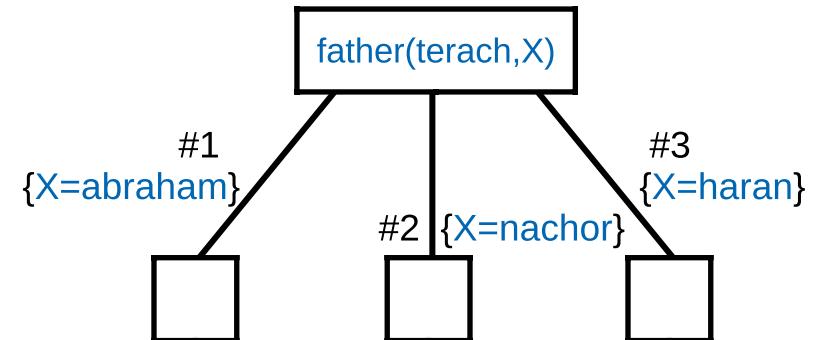
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Un *search tree* per la query `father(terach,X)`.

```
/* programma */  
/*1*/ father(terach, abraham).  
/*2*/ father(terach, nachor).  
/*3*/ father(terach, haran).  
/*4*/ father(abraham, isaac).  
    ...
```



- Gli archi corrispondono ai fatti che unificano con la query
- Sono etichettati con
 - ◆ il numero del fatto utilizzato (nell'ordine in cui compare nel programma)
 - ◆ il mgu della query e del fatto



Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

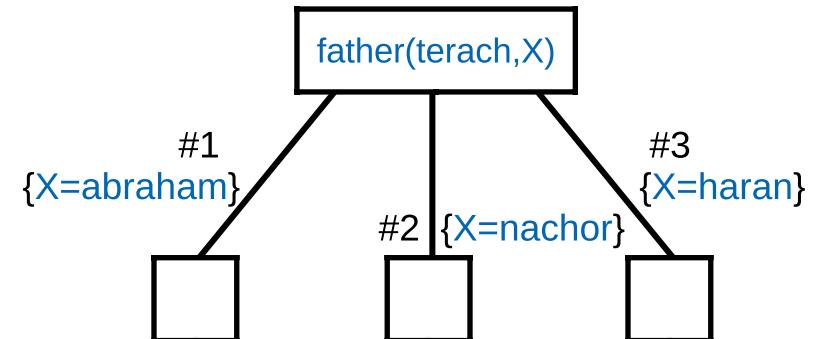
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Un *search tree* per la query `father(terach,X)`.

```
/* programma */  
/*1*/ father(terach, abraham).  
/*2*/ father(terach, nachor).  
/*3*/ father(terach, haran).  
/*4*/ father(abraham, isaac).  
    ...
```



- Gli archi corrispondono ai fatti che unificano con la query
- Sono etichettati con
 - ◆ il numero del fatto utilizzato (nell'ordine in cui compare nel programma)
 - ◆ il mgu della query e del fatto
- Il search tree di una query (rispetto a un programma) comprende *tutte* le risposte che Prolog genera se l'utente glielo chiede



Rappresentazione grafica del procedimento

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Sostituzioni

Istanze

L'unificazione

Search tree

Conjunctive queries

Conjunctive queries

Le Regole

Ragionamento

Liste

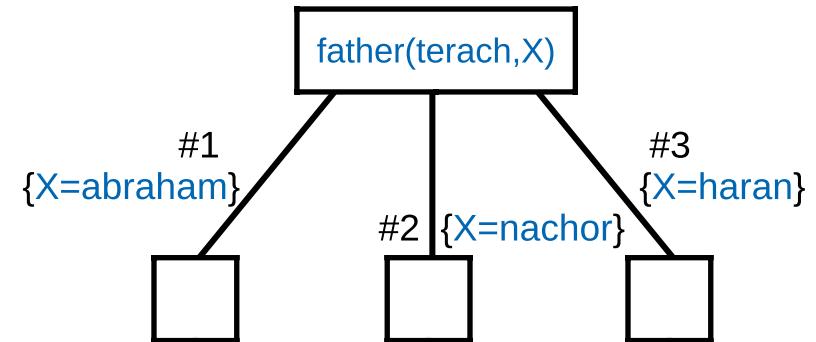
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Un *search tree* per la query `father(terach,X)`.

```
/* programma */  
/*1*/ father(terach, abraham).  
/*2*/ father(terach, nachor).  
/*3*/ father(terach, haran).  
/*4*/ father(abraham, isaac).  
    ...
```



- Gli archi corrispondono ai fatti che unificano con la query
- Sono etichettati con
 - ◆ il numero del fatto utilizzato (nell'ordine in cui compare nel programma)
 - ◆ il mgu della query e del fatto
- Il search tree di una query (rispetto a un programma) comprende *tutte* le risposte che Prolog genera se l'utente glielo chiede
- Le computazioni di Prolog corrispondono a una visita depth-first da sinistra a destra (ogni ramo di successo una risposta)



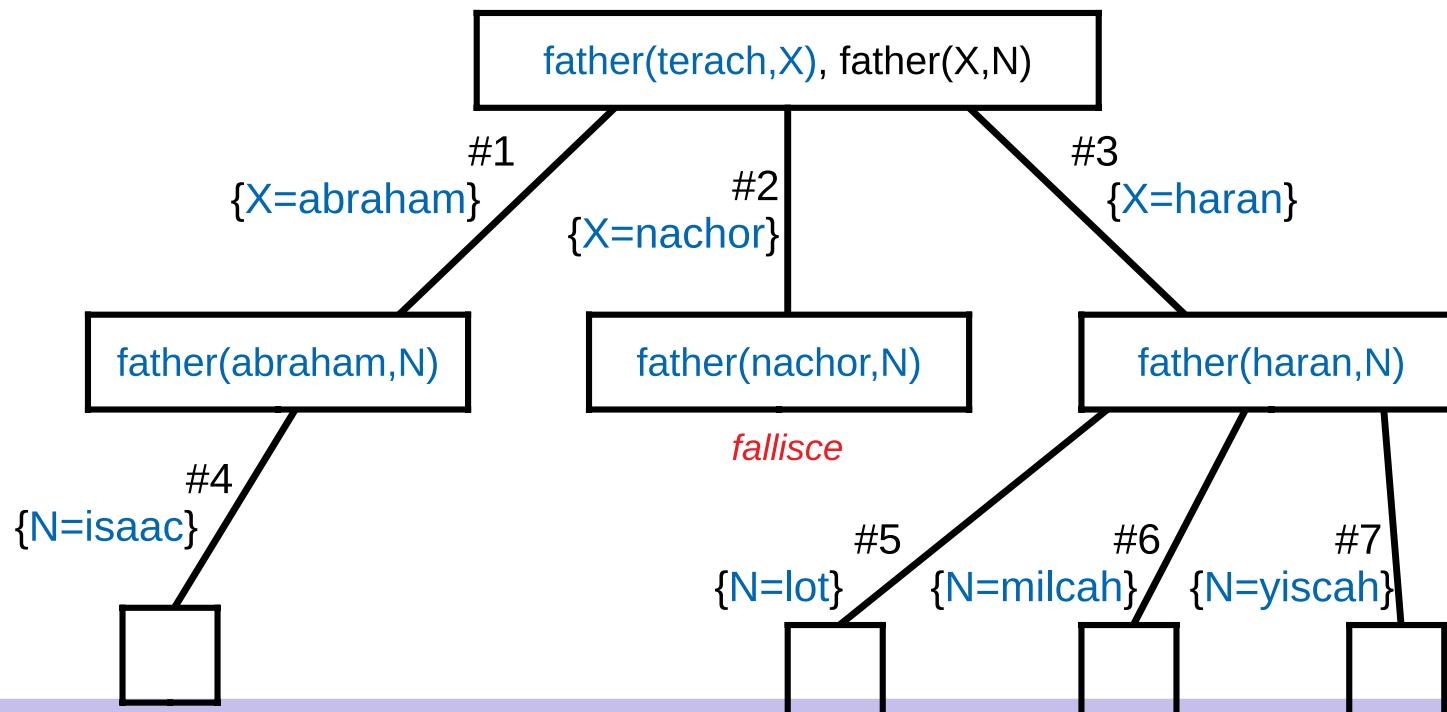
Conjunctive queries

- Le query possono contenere più formule atomiche (*goals*)
- Esempio: chi sono i nipoti di terach?

`father(terach,X), father(X,Nipote)`

(la virgola è un *and*) è come un *join*

```
/* programma */  
/*1*/ father(terach, abraham).  
/*2*/ father(terach, nachor).  
/*3*/ father(terach, haran).  
/*4*/ father(abraham, isaac).  
/*5*/ father(haran, lot).  
/*6*/ father(haran, milcah).  
/*7*/ father(haran, yiscah).  
...
```





Conjunctive queries

- Le figlie di haran le trovo con la query congiuntiva ... ?

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Conjunctive queries

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Conjunctive queries

- Le figlie di haran le trovo con la query congiuntiva ... ?

```
father(haran ,Y) , female(Y).
```

I valori di Y mi danno le figlie di haran

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Conjunctive queries

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Conjunctive queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Conjunctive queries

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

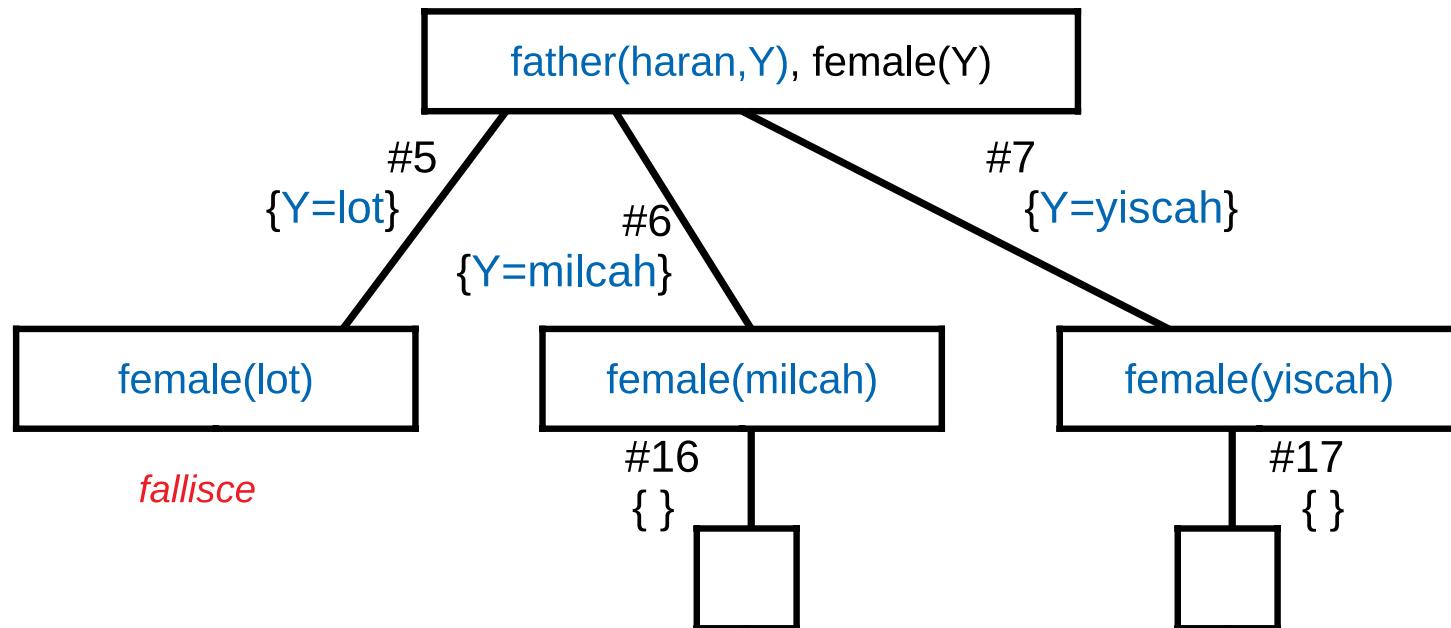
[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Le figlie di haran le trovo con la query congiuntiva ... ?

```
father(haran, Y), female(Y).
```

I valori di Y mi danno le figlie di haran



```
Y = milcah;
Y = yiscah;
false.
```



Conjunctive queries

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

Conjunctive queries

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

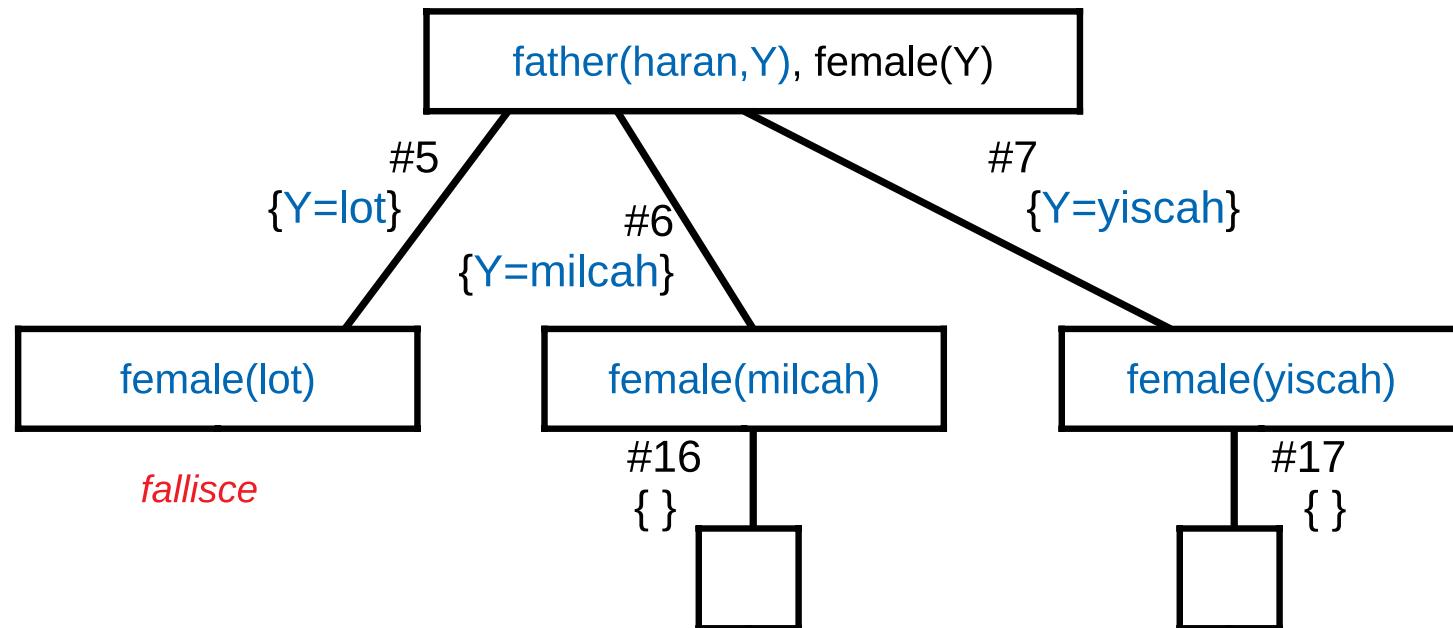
[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Le figlie di haran le trovo con la query congiuntiva ... ?

```
father(haran, Y), female(Y).
```

I valori di Y mi danno le figlie di haran



```
Y = milcah;
Y = yiscah;
false.
```

- Ma... perchè non definire il concetto di figlia nel programma?



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

Le Regole

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Le Regole



Le regole

- X è figlia di Y se Y è padre di X e X è femmina

```
daughter(X,Y) :- father(Y,X), female(X).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- X è figlia di Y se Y è padre di X e X è femmina

```
daughter(X,Y) :- father(Y,X), female(X).
```

- In generale la sintassi delle regole è

```
<rule> ::= <atomic formula> [:- <goal list>].
```

```
<goal list> ::= <atomic formula> [, <atomic formula>]*
```



Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- X è figlia di Y se Y è padre di X e X è femmina

```
daughter(X,Y) :- father(Y,X), female(X).
```

- In generale la sintassi delle regole è

```
<rule> ::= <atomic formula> [:- <goal list>].
```

```
<goal list> ::= <atomic formula> [, <atomic formula>]*
```

- ◆ $:-$ sta per \Leftarrow
- ◆ la formula atomica prima di $:-$ è detta **testa (head)**



Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- X è figlia di Y se Y è padre di X e X è femmina

```
daughter(X,Y) :- father(Y,X), female(X).
```

- In generale la sintassi delle regole è

```
<rule> ::= <atomic formula> [:- <goal list>].
```

```
<goal list> ::= <atomic formula> [, <atomic formula>]*
```

- ◆ :- sta per \Leftarrow
- ◆ la formula atomica prima di :- è detta **testa (head)**
- ◆ la parte dopo :- è detta **corpo (body)**



Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- X è figlia di Y se Y è padre di X e X è femmina

```
daughter(X,Y) :- father(Y,X), female(X).
```

- In generale la sintassi delle regole è

```
<rule> ::= <atomic formula> [:- <goal list>].
```

```
<goal list> ::= <atomic formula> [, <atomic formula>]*
```

- ◆ `:-` sta per \Leftarrow
- ◆ la formula atomica prima di `:-` è detta **testa (head)**
- ◆ la parte dopo `:-` è detta **corpo (body)**
- ◆ notare che i fatti non sono che regole senza corpo



Le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- X è figlia di Y se Y è padre di X e X è femmina

```
daughter(X,Y) :- father(Y,X), female(X).
```

- In generale la sintassi delle regole è

```
<rule> ::= <atomic formula> [:- <goal list>].  
  
<goal list> ::= <atomic formula> [, <atomic formula>]*
```

- ◆ $:$ - sta per \Leftarrow
- ◆ la formula atomica prima di $:$ - è detta **testa (head)**
- ◆ la parte dopo $:$ - è detta **corpo (body)**
- ◆ notare che i fatti non sono che regole senza corpo

- **Definizione:** un programma logico è un insieme di regole



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[**Ragionamento**](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

Programmazione

nondeterministica

[Unicità di Prolog](#)

Ragionamento



Ragionare con le regole

- Aggiungiamo in coda al programma (posizione 18) la regola

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Z, haran)` si trovano così:

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Z, haran)` si trovano così:

```
daughter(Z,haran)
```

Cerca una testa che unifica con la query
e sostituisce la query col corpo
istanziato con il mgu



Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

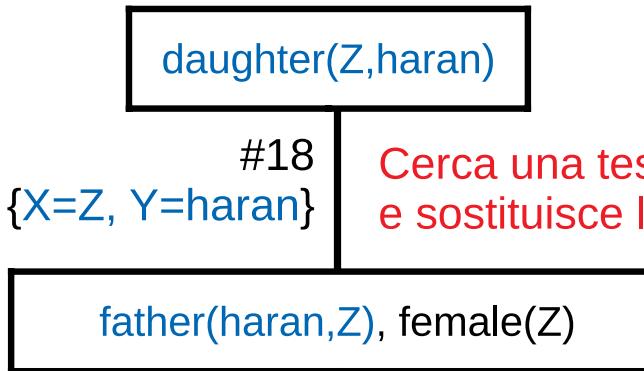
Programmazione nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Z, haran)` si trovano così:



Cerca una testa che unifica con la query
e sostituisce la query col corpo
istanziato con il mgu



Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

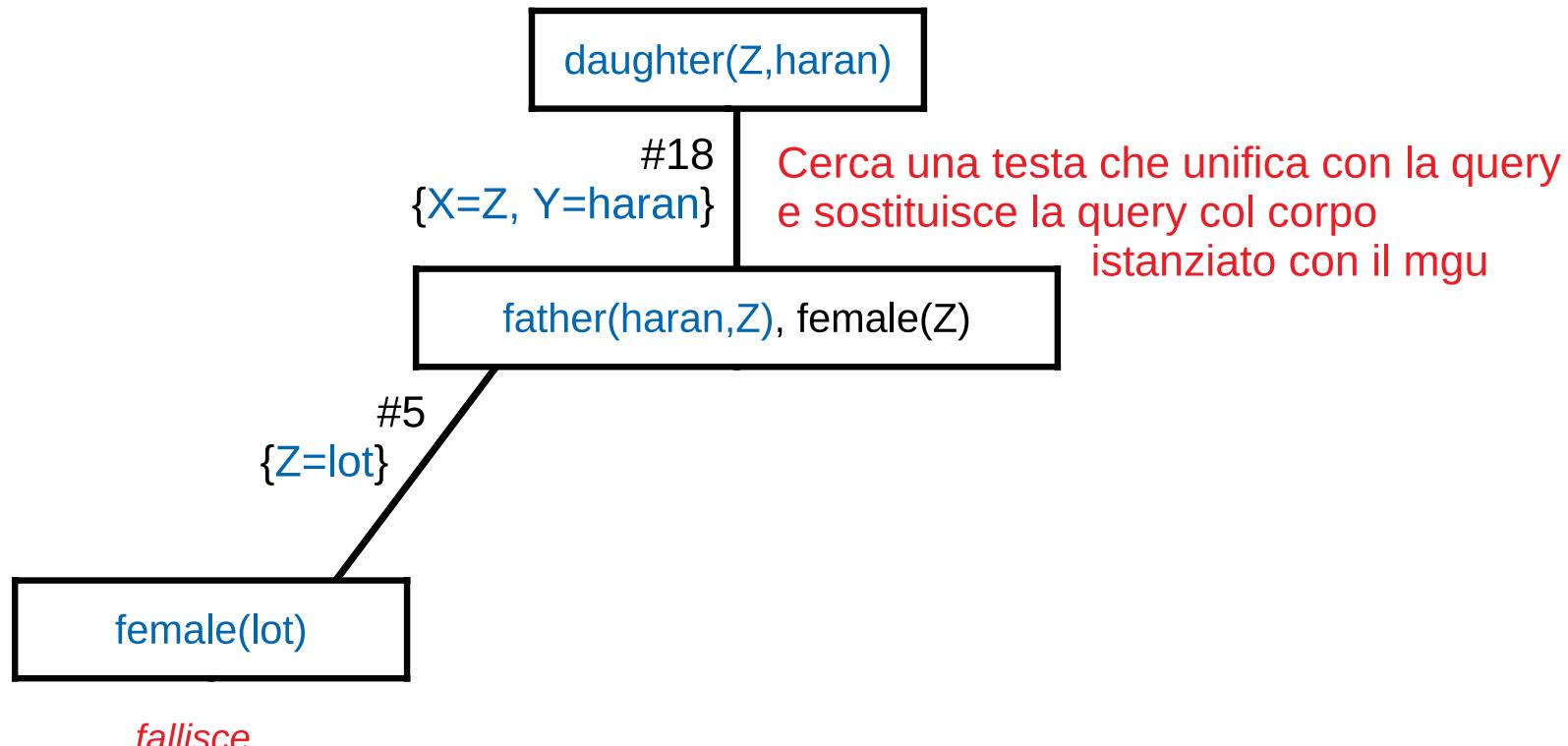
Programmazione
nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Z, haran)` si trovano così:





Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

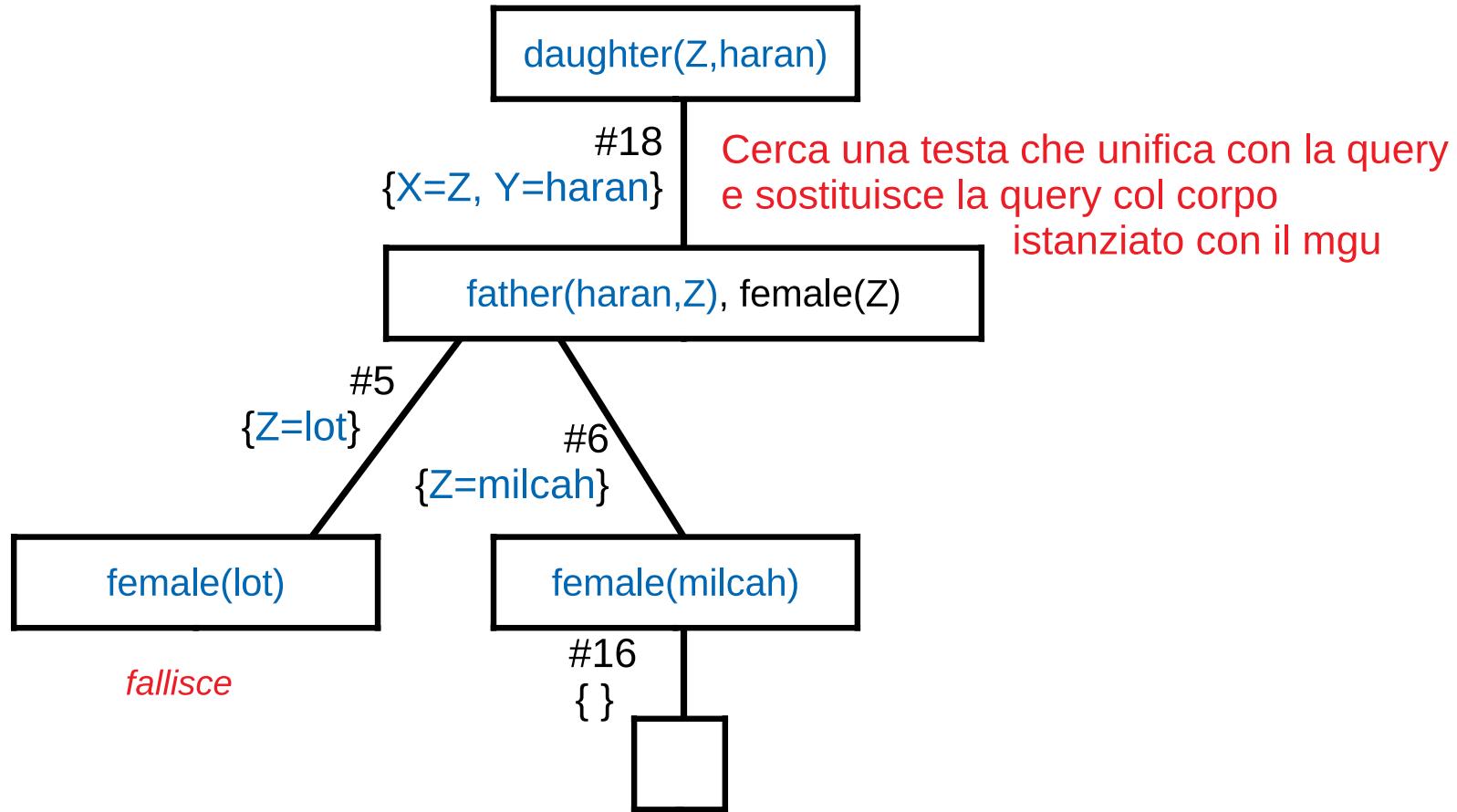
Programmazione
nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Z, haran)` si trovano così:





Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

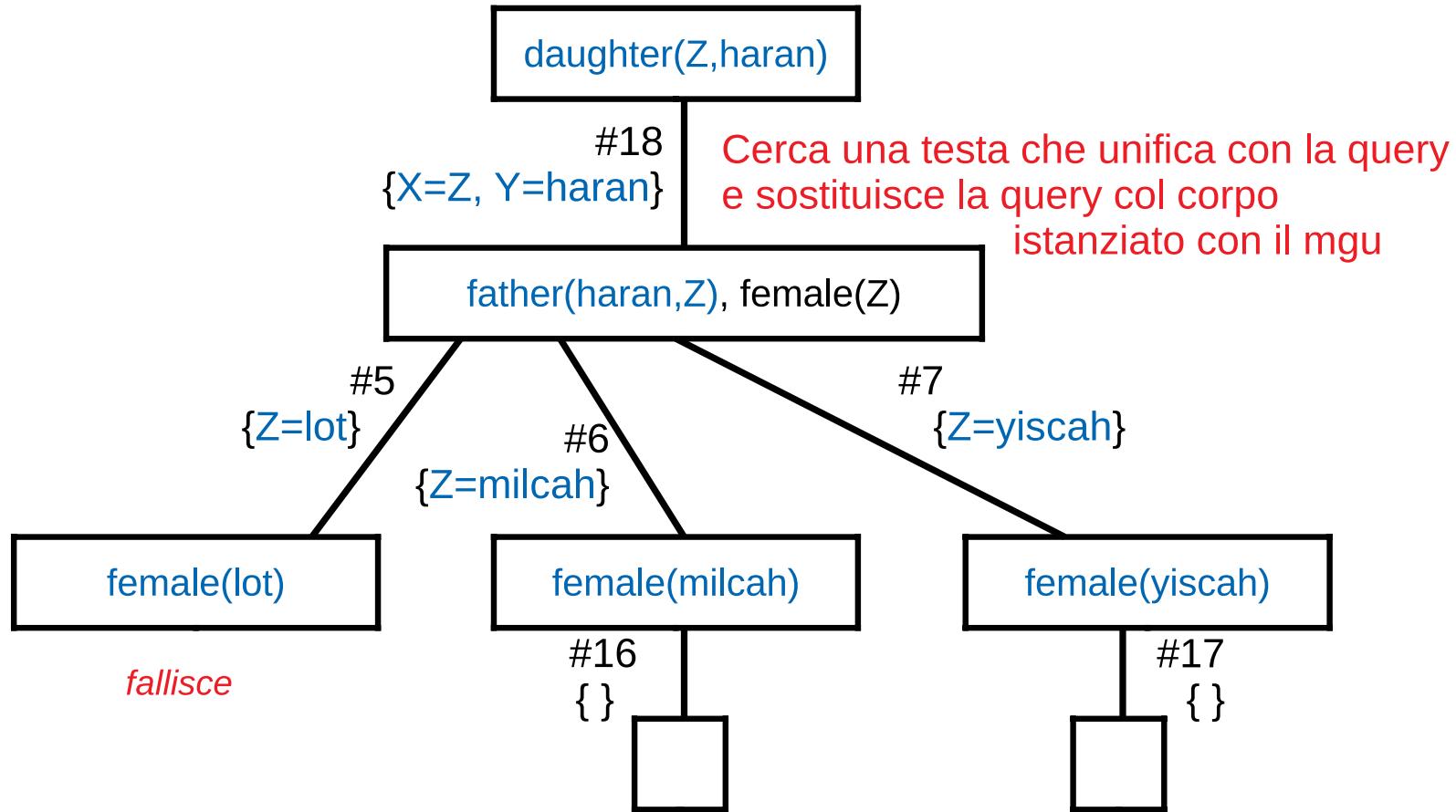
Programmazione nondeterministica

Unicità di Prolog

- Aggiungiamo in coda al programma (posizione 18) la regola

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Z, haran)` si trovano così:





Ragionare con le regole

- Bisogna fare attenzione con le variabili
 - ◆ quelle della query devono sempre essere diverse da quelle della regola

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Ragionare con le regole

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

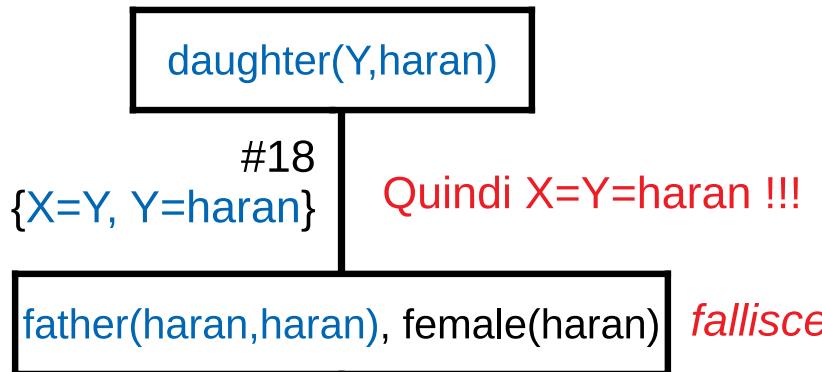
Bisogna fare attenzione con le variabili

- ◆ quelle della query devono sempre essere diverse da quelle della regola

Esempio di problema se non fosse così:

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Y, haran)` verrebbero perse





Ragionare con le regole

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

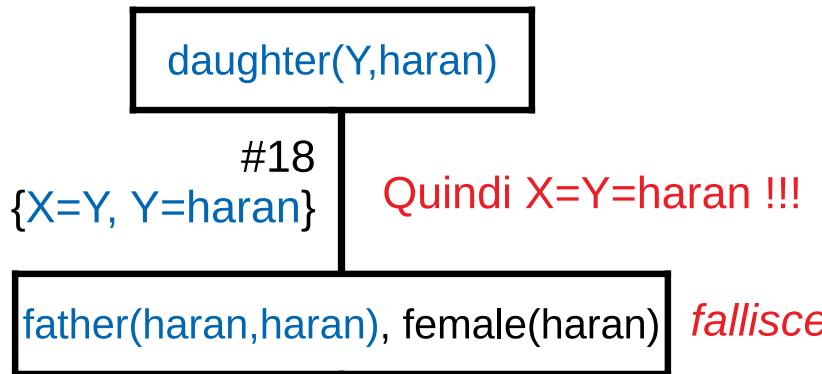
- Bisogna fare attenzione con le variabili

- ◆ quelle della query devono sempre essere diverse da quelle della regola

Esempio di problema se non fosse così:

```
daughter(X,Y) :- father(Y,X), female(X).
```

Le risposte alla query `daughter(Y, haran)` verrebbero perse



- Per evitare questo problema, ogni volta che una regola viene usata la WAM ridenomina le sue variabili, ad esempio

```
daughter(_101,_102) :- father(_102,_101), female(_101).
```



Migliorando l'esempio

- Con la definizione attuale ci perdiamo le figlie delle donne
- **Soluzione 1:** aggiungere un'altra regola in fondo al programma

```
daughter(X, Y) :- mother(Y, X), female(X).
```

che mi permette di dedurre che X è figlia di Y quando Y è madre di X e X è femmina

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Migliorando l'esempio

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Con la definizione attuale ci perdiamo le figlie delle donne
- **Soluzione 1:** aggiungere un'altra regola in fondo al programma

```
daughter(X, Y) :- mother(Y, X), female(X).
```

che mi permette di dedurre che X è figlia di Y quando Y è madre di X e X è femmina

- **Soluzione 2:** introdurre il concetto di *genitore* (*parent*)

```
parent(X, Y) :- father(X, Y).  
parent(X, Y) :- mother(X, Y).
```



Migliorando l'esempio

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- Con la definizione attuale ci perdiamo le figlie delle donne
- **Soluzione 1:** aggiungere un'altra regola in fondo al programma

```
daughter(X,Y) :- mother(Y,X), female(X).
```

che mi permette di dedurre che X è figlia di Y quando Y è madre di X e X è femmina

- **Soluzione 2:** introdurre il concetto di *genitore* (parent)

```
parent(X,Y) :- father(X,Y).  
parent(X,Y) :- mother(X,Y).  
/* puo' essere riutilizzato per diverse definizioni */  
  
daughter(X,Y) :- parent(Y,X), female(X).  
  
son(X,Y) :- parent(Y,X), male(X).  
  
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```



Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```



Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

Programmazione
nondeterministica

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```

Query:

son(isaac,Z)



Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

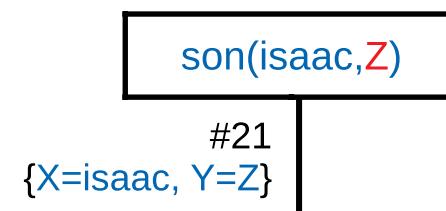
[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

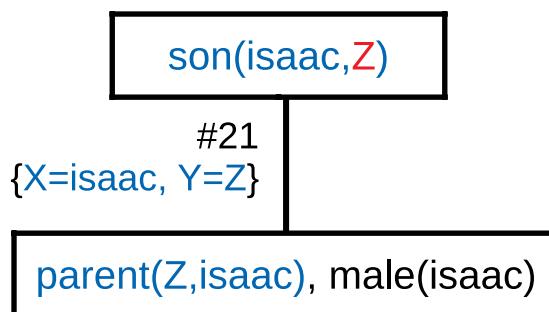
[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

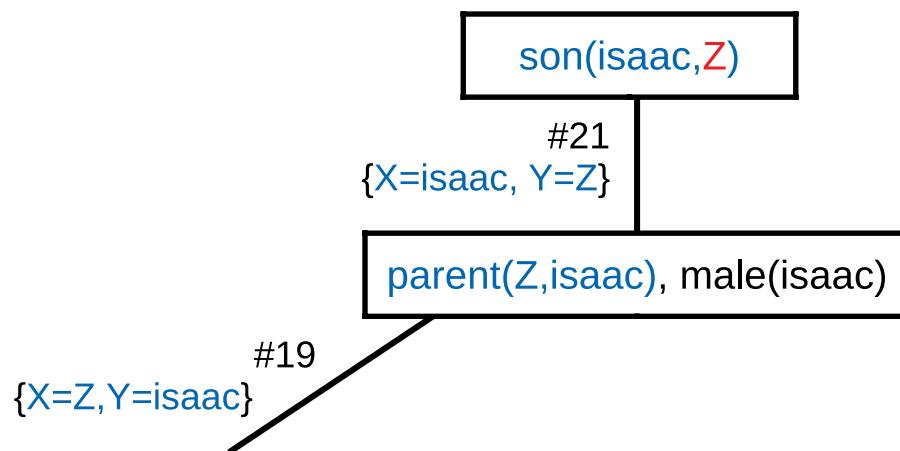
[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

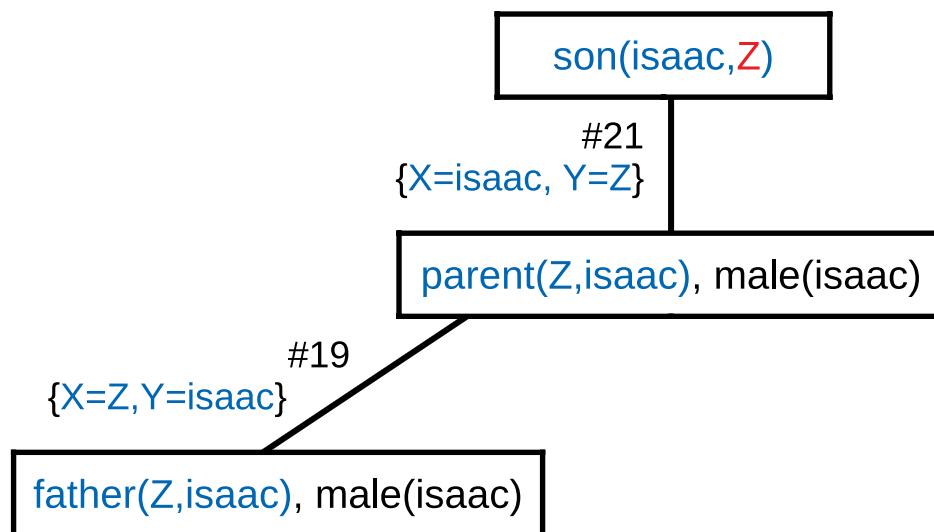
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

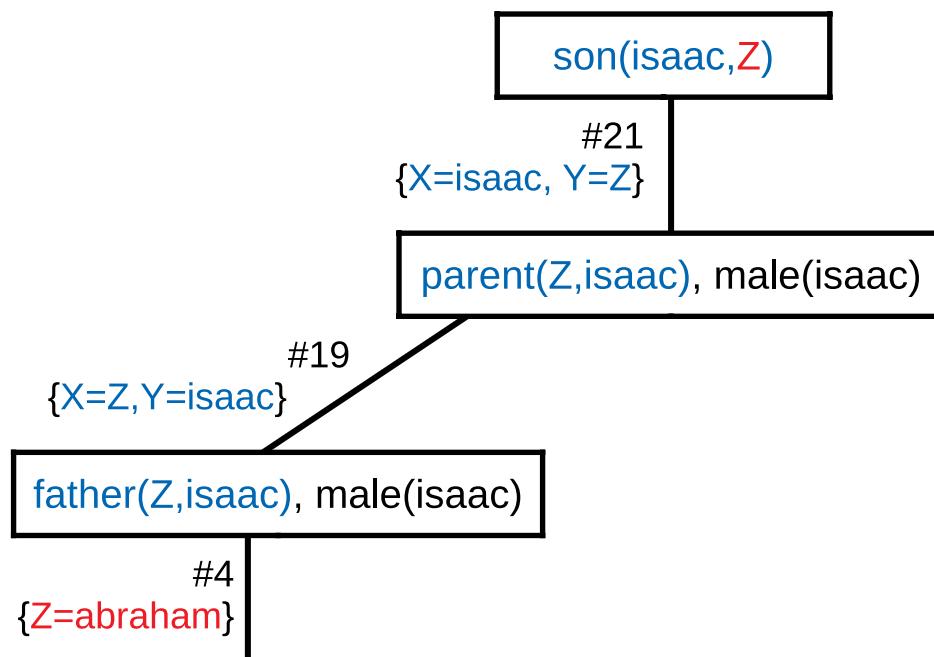
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

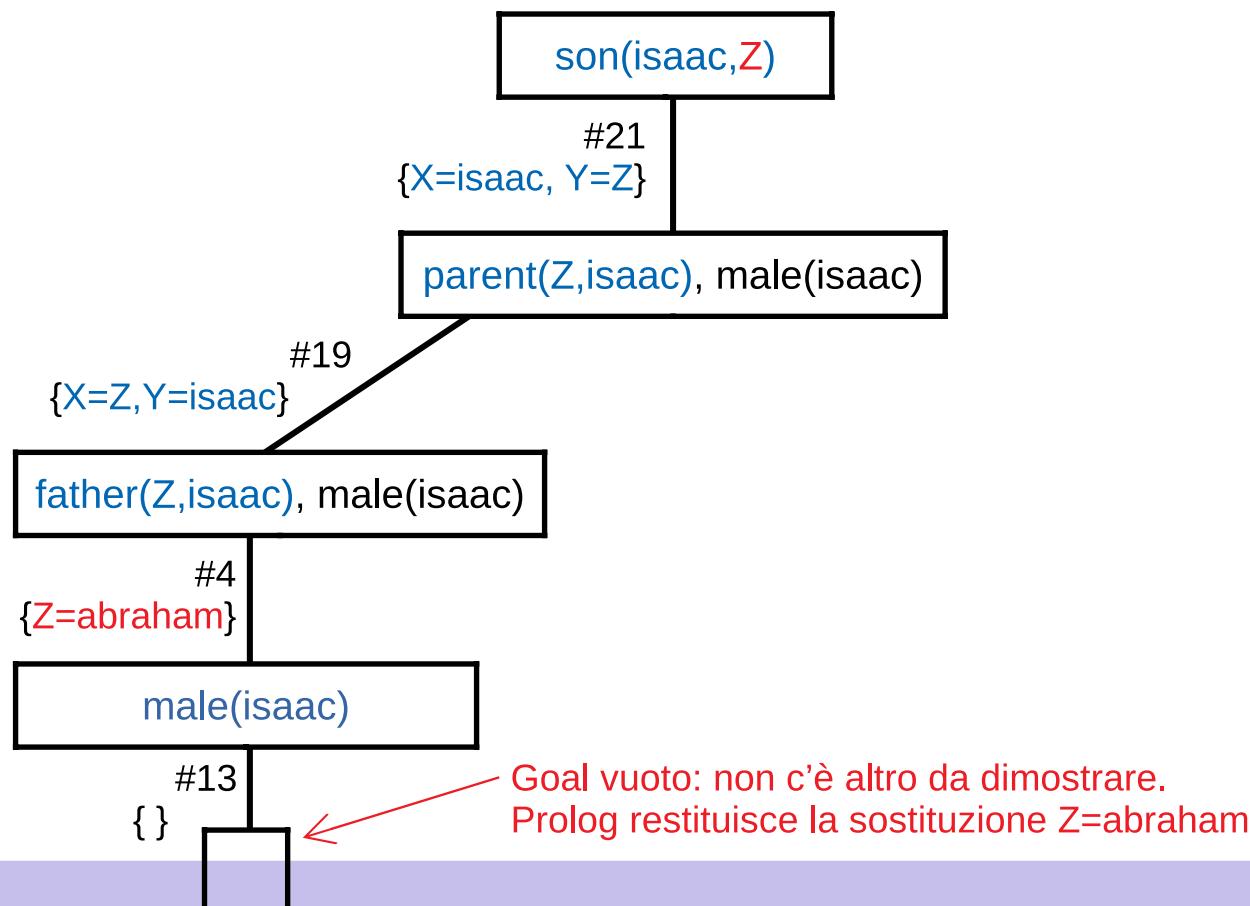
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19 */ parent(X,Y) :- father(X,Y).  
/*20 */ parent(X,Y) :- mother(X,Y).  
/*21 */ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

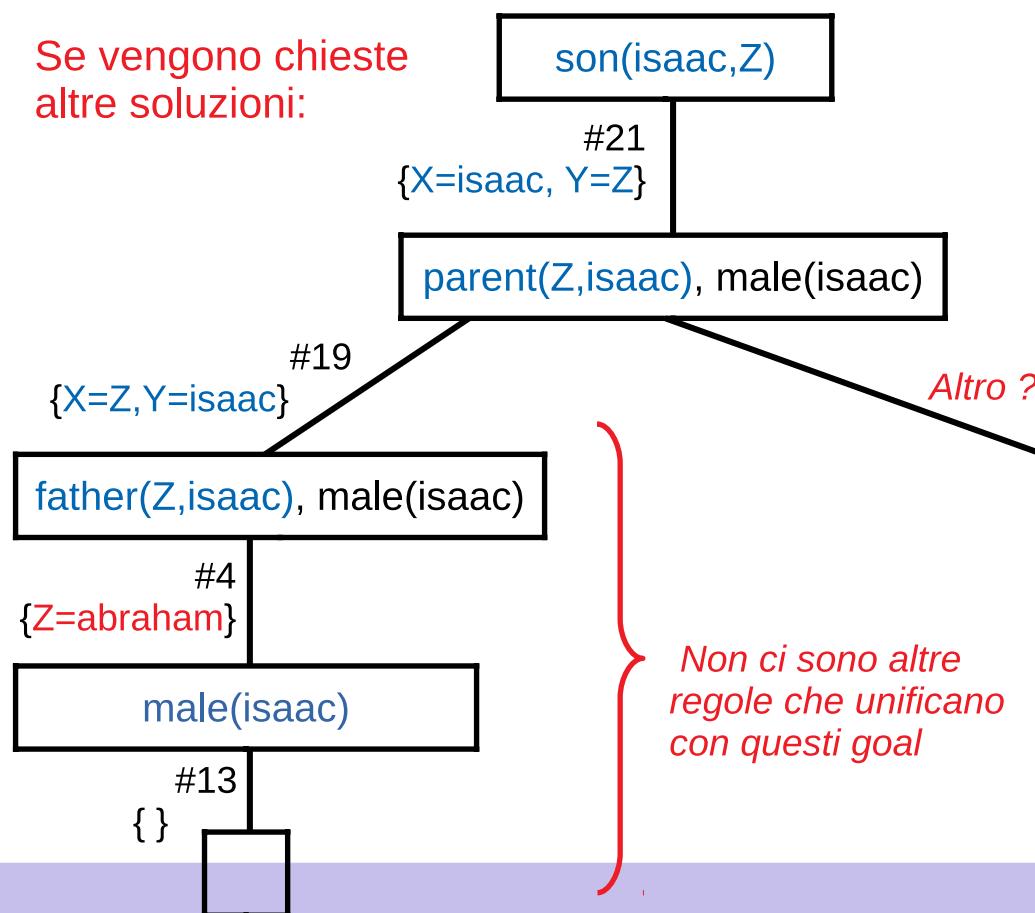
Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19 */ parent(X,Y) :- father(X,Y).  
/*20 */ parent(X,Y) :- mother(X,Y).  
/*21 */ son(X,Y)      :- parent(Y,X), male(X).
```

Se vengono chieste
altre soluzioni:





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

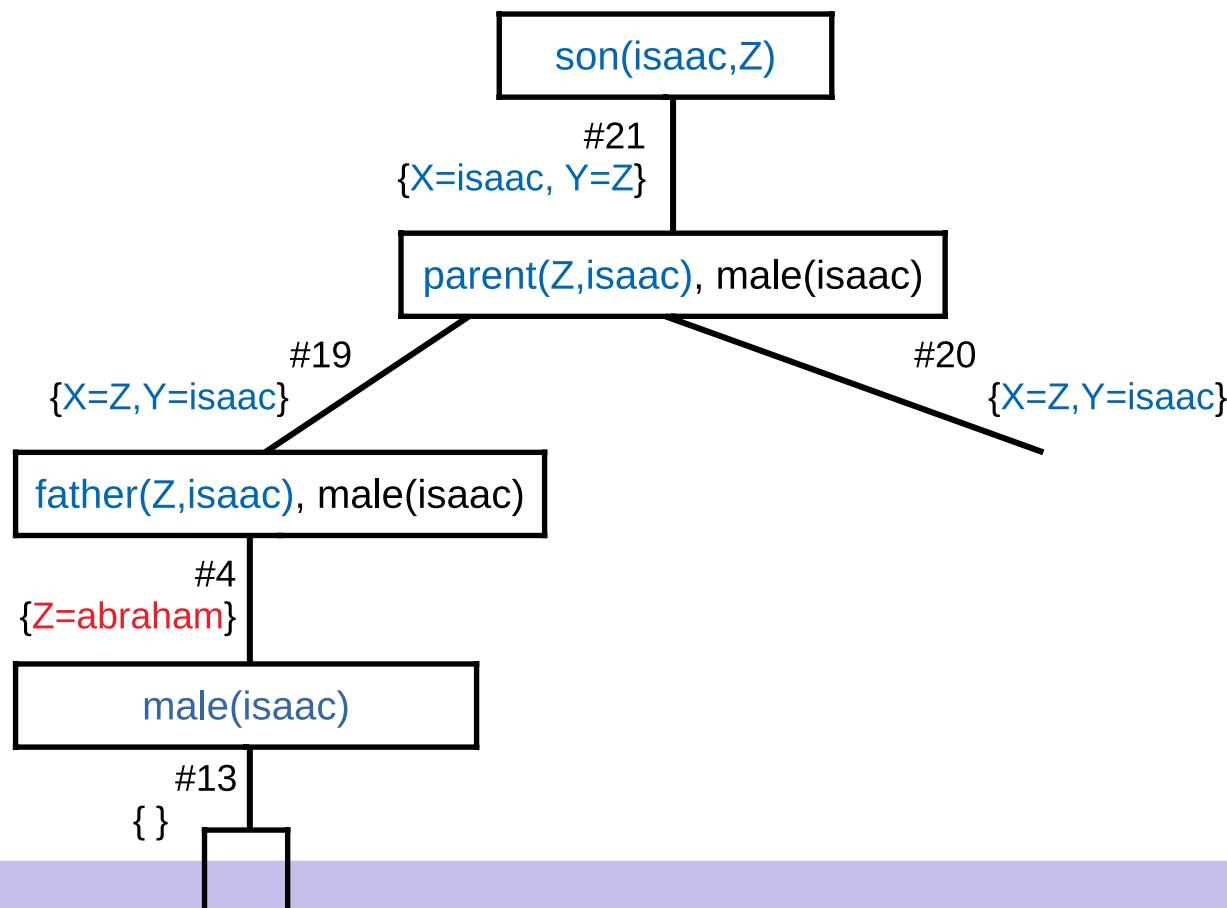
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

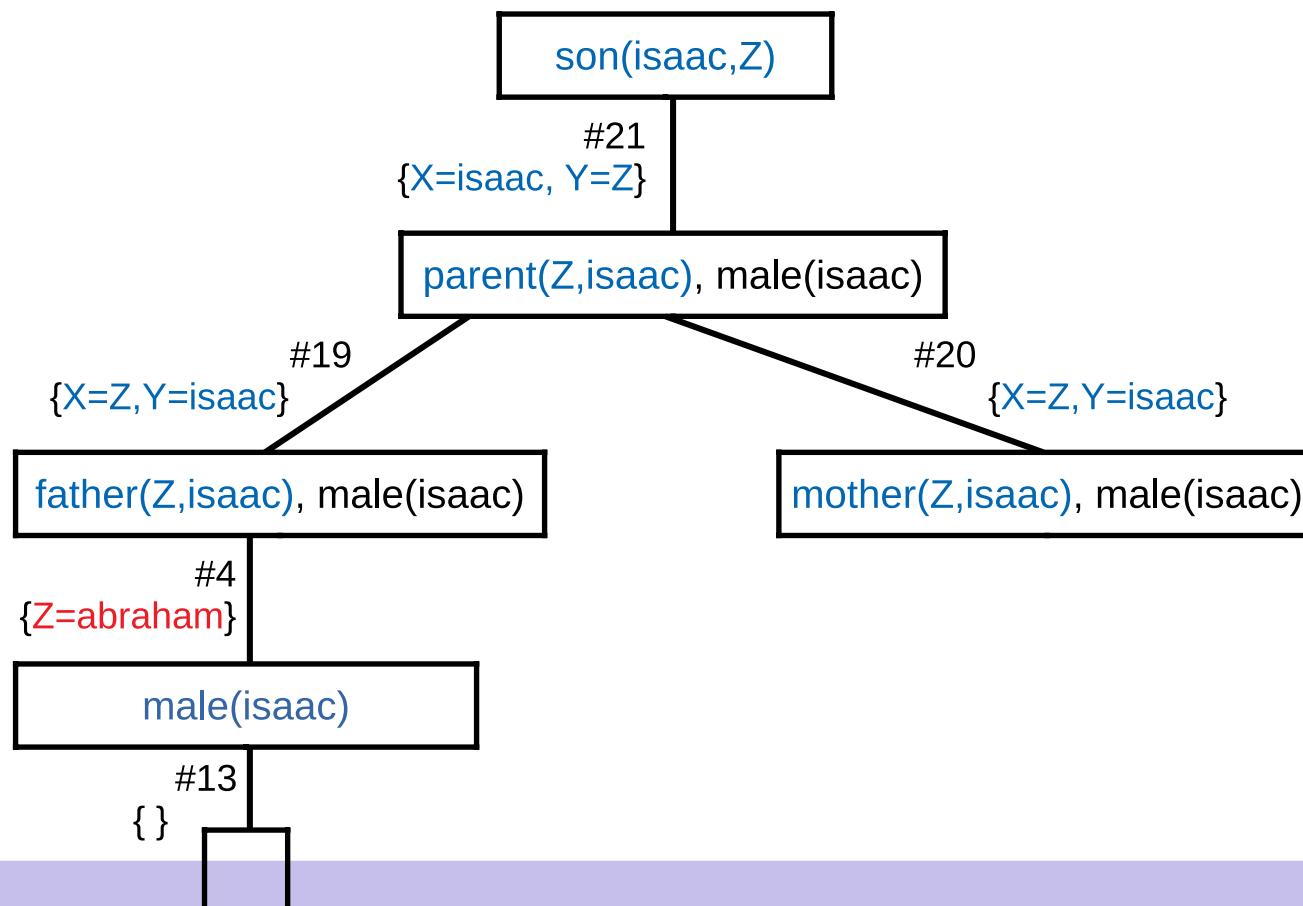
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

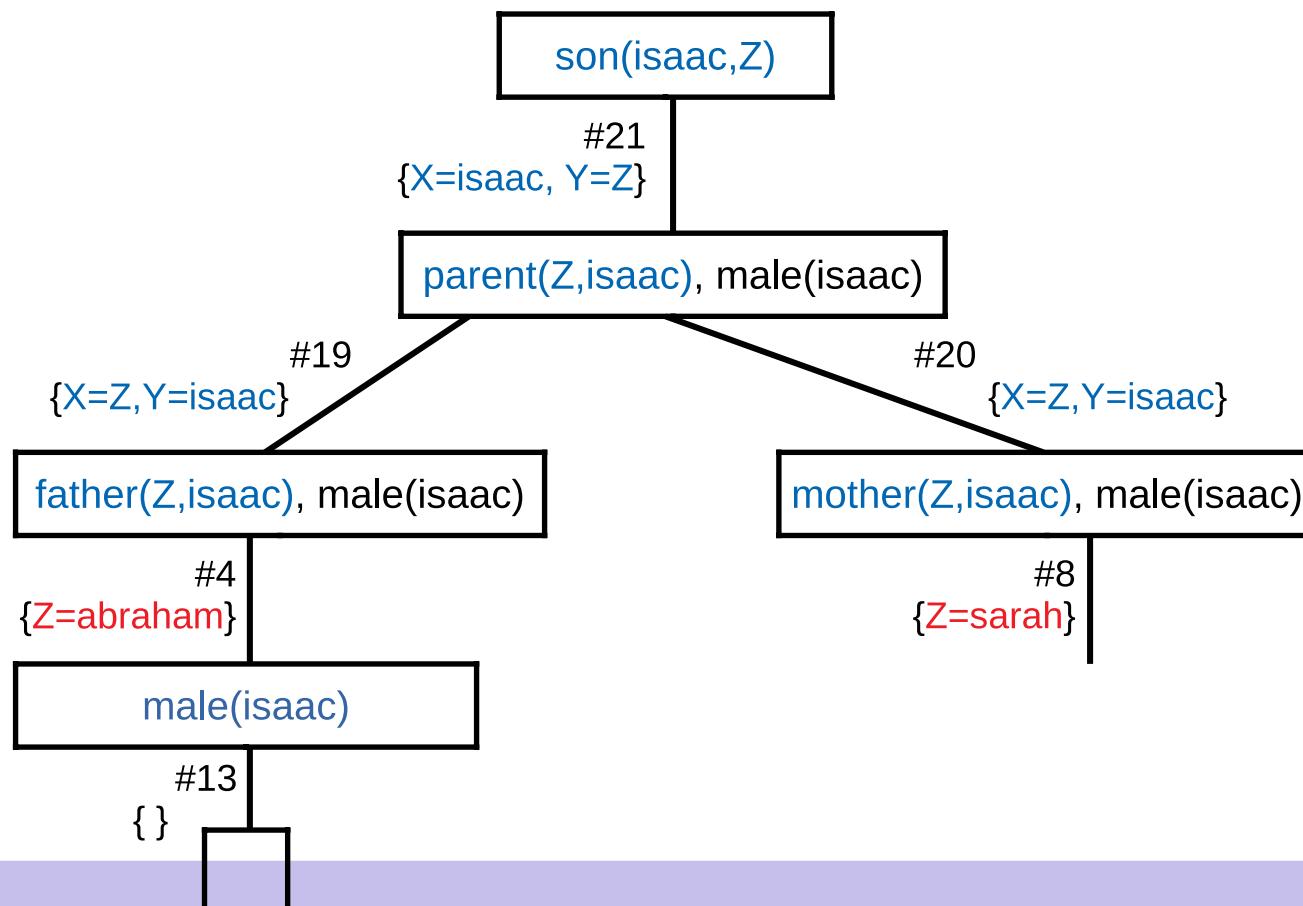
[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

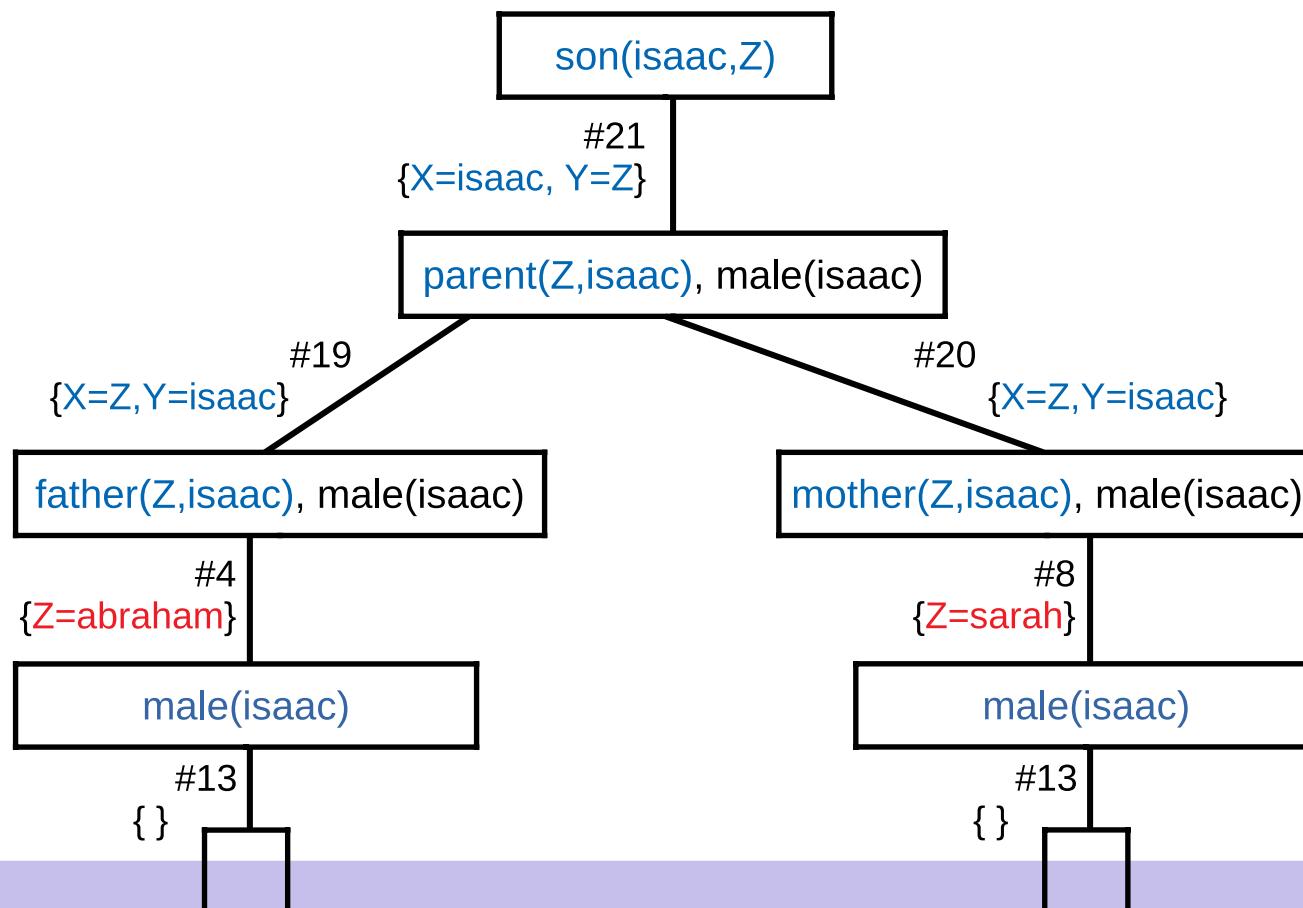
Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19*/ parent(X,Y) :- father(X,Y).  
/*20*/ parent(X,Y) :- mother(X,Y).  
/*21*/ son(X,Y)      :- parent(Y,X), male(X).
```





Esempio di derivazione con standardization apart

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

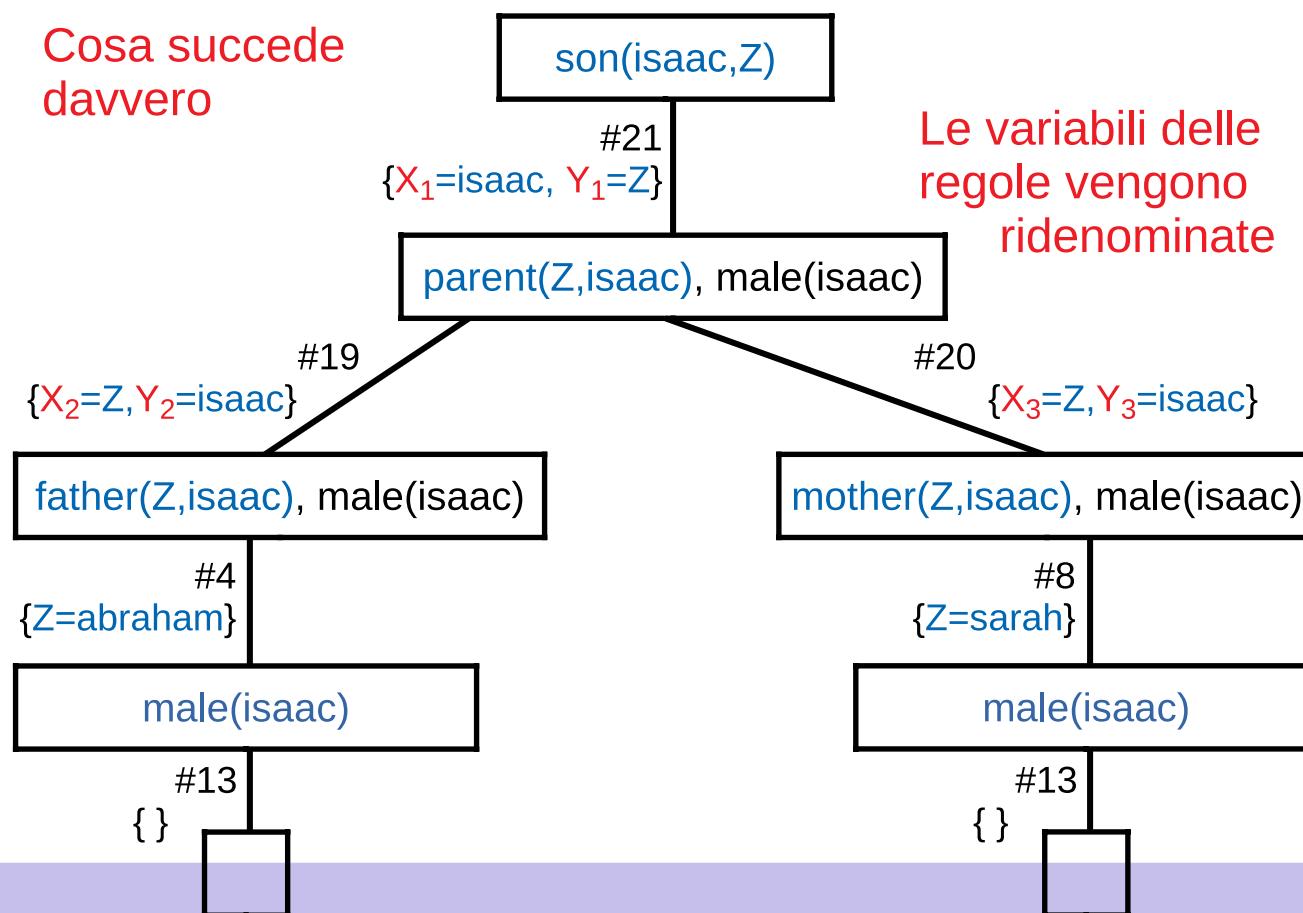
[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

```
/* 4 */ father(abraham,isaac).      ....  
/* 8 */ mother(sarah,isaac).       ....  
/*13*/ male(isaac).             ....  
/*19 */ parent(X,Y) :- father(X,Y).  
/*20 */ parent(X,Y) :- mother(X,Y).  
/*21 */ son(X,Y)      :- parent(Y,X), male(X).
```

Cosa succede
davvero



Le variabili delle
regole vengono
ridenominate



Overloading

- Si può usare lo stesso nome di predicato con numeri diversi di argomenti
- Simili prediciati vengono trattati come prediciati diversi
- Esempio: dall'informazione che “X è madre di Y” derivare il concetto di essere madre

```
mother(Mom) :- mother(Mom,X).  
/* Mom è una mamma se esiste un X tale che Mom è madre di X */  
/* Notare che abbiamo un mother unario e uno binario */
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Wildcards

- Notare che nel goal `mother(Mom, X)` il valore di `X` non interessa
- In questi casi possiamo usare wildcards simili a ML

```
mother(Mom) :- mother(Mom, _).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

Wildcards

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Wildcards

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Notare che nel goal `mother(Mom, X)` il valore di `X` non interessa
- In questi casi possiamo usare wildcards simili a ML

```
mother(Mom) :- mother(Mom, _).
```

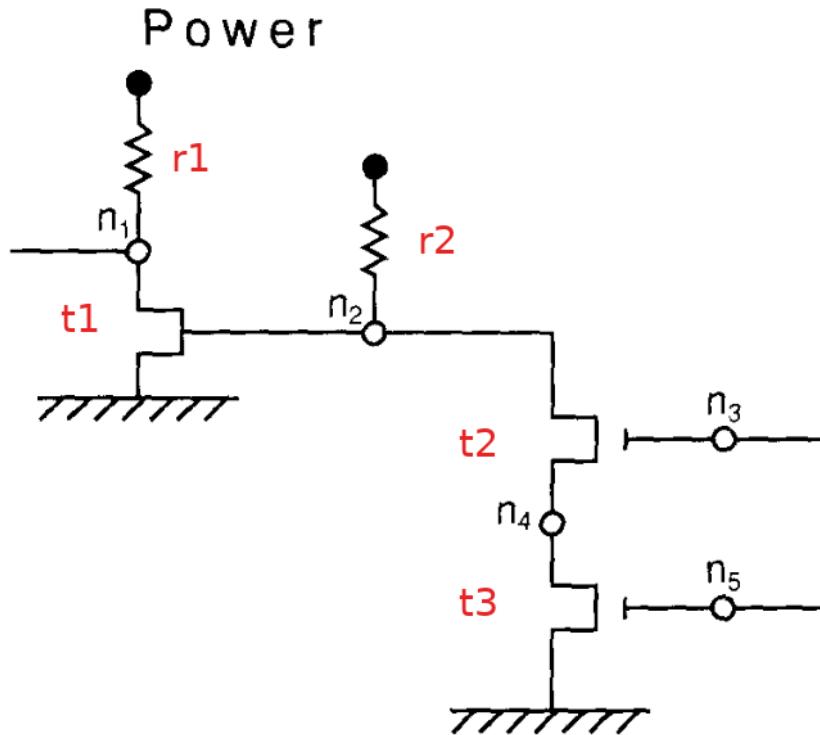
- Attenzione: se usiamo due volte una variabile (ad es. `X`) allora in quei punti devo avere lo stesso valore, mentre ogni wildcard può essere associata a un valore diverso

```
?- mother(X, X).  
false      (nessuna è madre di sè stessa)  
  
?- mother(_, _).  
true       (cerca un fatto mother(X, Y) nel database)
```



Esempio: Analisi di Circuiti

Con i fatti possiamo descrivere circuiti



```
resistor(r1, power, n1).  
resistor(r2, power, n2).  
transistor(t1, n2, ground, n1).  
transistor(t2, n3, n4, n2).  
transistor(t3, n5, ground, n4).
```

Figure 2.2 A logical circuit

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog



Esempio: Analisi di Circuiti

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

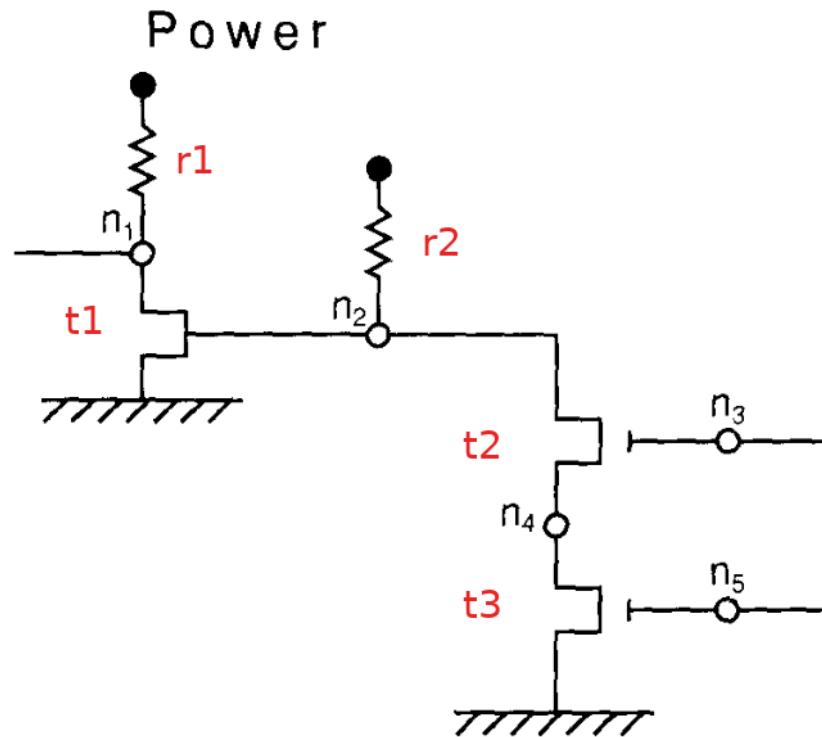
Liste

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

Con i fatti possiamo descrivere circuiti



```
resistor(r1, power, n1).  
resistor(r2, power, n2).  
transistor(t1, n2, ground, n1).  
transistor(t2, n3, n4, n2).  
transistor(t3, n5, ground, n4).
```

Figure 2.2 A logical circuit

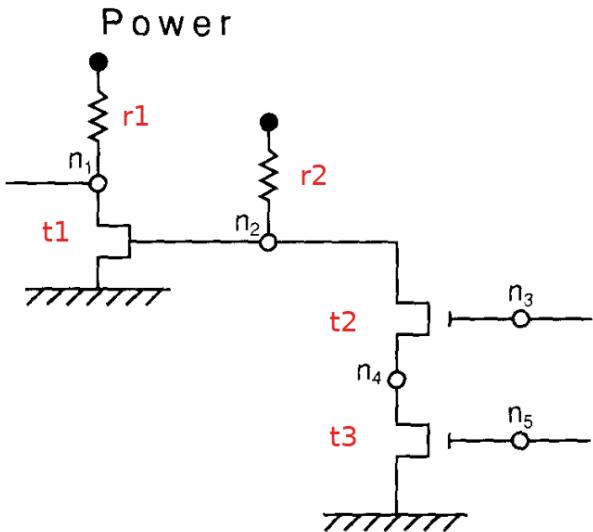
Da questa descrizione fisica vogliamo derivarne una funzionale

- ovvero: cosa fanno questi componenti?



Esempio: Analisi di Circuiti

■ Descrizione funzionale dei circuiti



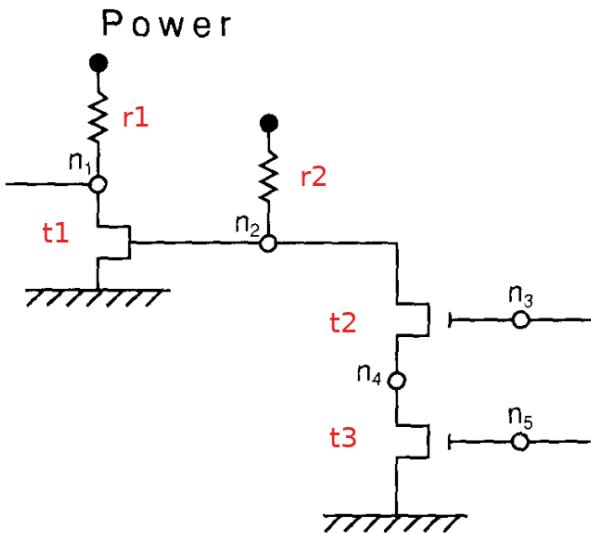
```
inverter( inv(T,R), In, Out ) :-  
    transistor(T, In, ground, Out ),  
    resistor(R, power, Out ).
```

Figure 2.2 A logical circuit



Esempio: Analisi di Circuiti

■ Descrizione funzionale dei circuiti



```
inverter( inv(T,R), In, Out ) :-  
    transistor(T, In, ground, Out ),  
    resistor(R, power, Out ).  
  
nand_gate( nand(T1,T2,R), In1, In2, Out ) :-  
    transistor(T1, In1, X, Out),  
    transistor(T2, In2, ground, X),  
    resistor(R,power,Out).
```

Figure 2.2 A logical circuit



Esempio: Analisi di Circuiti

■ Descrizione funzionale dei circuiti

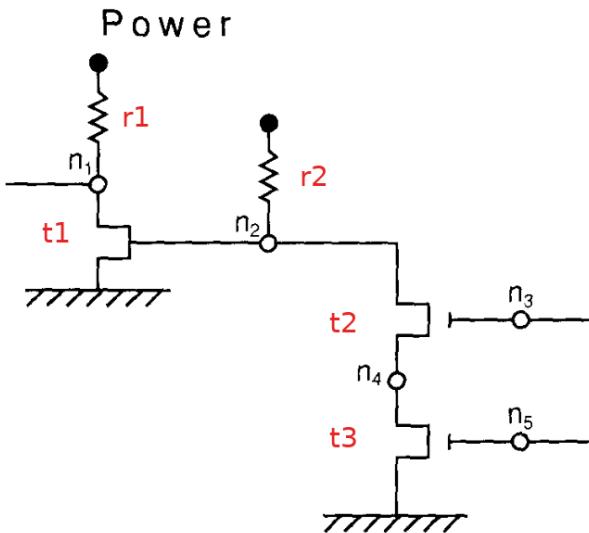


Figure 2.2 A logical circuit

```
inverter( inv(T,R), In, Out ) :-  
    transistor(T, In, ground, Out ),  
    resistor(R, power, Out ).  
  
nand_gate( nand(T1,T2,R), In1, In2, Out ) :-  
    transistor(T1, In1, X, Out),  
    transistor(T2, In2, ground, X),  
    resistor(R,power,Out).  
  
and_gate( and(N,I), In1, In2, Out ) :-  
    nand_gate(N, In1, In2, X),  
    inverter(I,X,Out).
```



Esempio: Analisi di Circuiti

■ Descrizione funzionale dei circuiti

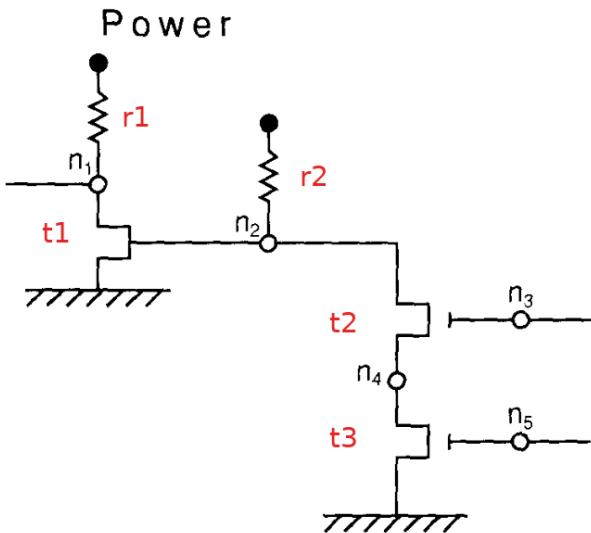


Figure 2.2 A logical circuit

```
inverter( inv(T,R), In, Out ) :-  
    transistor(T, In, ground, Out ),  
    resistor(R, power, Out ).  
  
nand_gate( nand(T1,T2,R), In1, In2, Out ) :-  
    transistor(T1, In1, X, Out),  
    transistor(T2, In2, ground, X),  
    resistor(R,power,Out).  
  
and_gate( and(N,I), In1, In2, Out ) :-  
    nand_gate(N, In1, In2, X),  
    inverter(I,X,Out).
```

Ora possiamo trovare i componenti logici implementati dal circuito

```
?- inverter(X,Y,Z).  
X = inv(t1, r1),  
Y = n2,  
Z = n1.
```



Esempio: Analisi di Circuiti

■ Descrizione funzionale dei circuiti

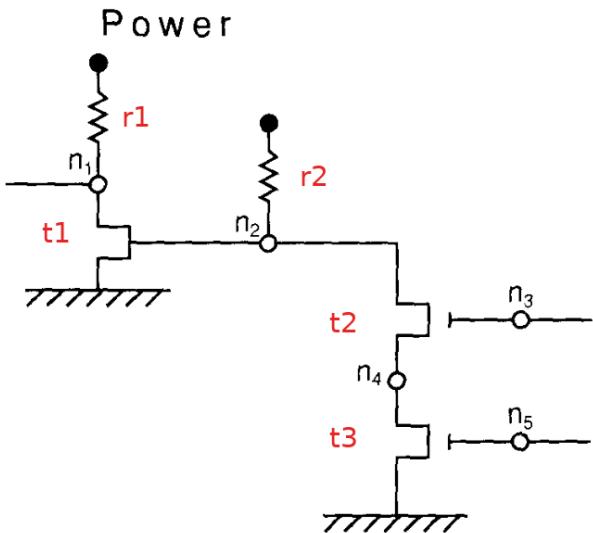


Figure 2.2 A logical circuit

```
inverter( inv(T,R), In, Out ) :-  
    transistor(T, In, ground, Out ),  
    resistor(R, power, Out ).  
  
nand_gate( nand(T1,T2,R), In1, In2, Out ) :-  
    transistor(T1, In1, X, Out),  
    transistor(T2, In2, ground, X),  
    resistor(R,power,Out).  
  
and_gate( and(N,I), In1, In2, Out ) :-  
    nand_gate(N, In1, In2, X),  
    inverter(I,X,Out).
```

Ora possiamo trovare i componenti logici implementati dal circuito

```
?- inverter(X,Y,Z).  
X = inv(t1, r1),  
Y = n2,  
Z = n1.
```

```
?- nand_gate(X,Y,Z,0).  
X = nand(t2, t3, r2),  
Y = n3,  
Z = n5,  
0 = n2.
```



Esempio: Analisi di Circuiti

■ Descrizione funzionale dei circuiti

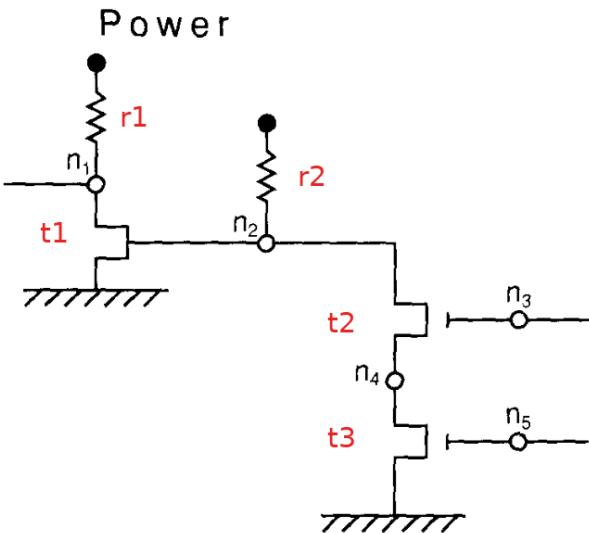


Figure 2.2 A logical circuit

```
inverter( inv(T,R), In, Out ) :-  
    transistor(T, In, ground, Out ),  
    resistor(R, power, Out ).  
  
nand_gate( nand(T1,T2,R), In1, In2, Out ) :-  
    transistor(T1, In1, X, Out),  
    transistor(T2, In2, ground, X),  
    resistor(R,power,Out).  
  
and_gate( and(N,I), In1, In2, Out ) :-  
    nand_gate(N, In1, In2, X),  
    inverter(I,X,Out).
```

Ora possiamo trovare i componenti logici implementati dal circuito

```
?- inverter(X,Y,Z).  
X = inv(t1, r1),  
Y = n2,  
Z = n1.
```

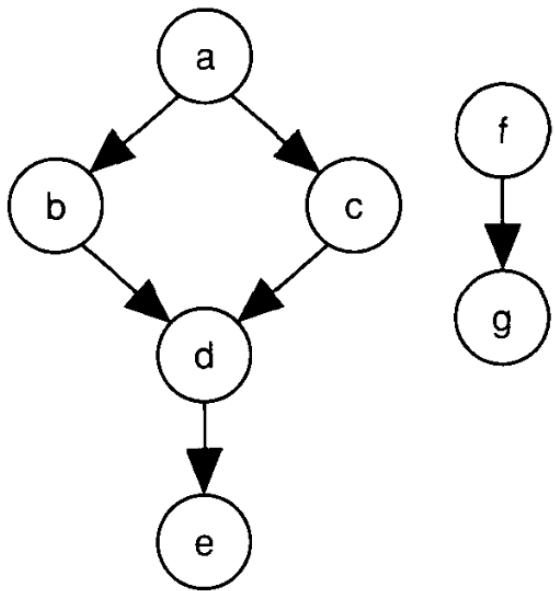
```
?- nand_gate(X,Y,Z,0).  
X = nand(t2, t3, r2),  
Y = n3,  
Z = n5,  
0 = n2.
```

```
?- and_gate(X,Y,Z,0).  
X =  
    and(nand(t2,t3,r2),inv(t1,r1)),  
Y = n3,  
Z = n5,  
0 = n1.
```



Regole ricorsive

Cerchiamo ora i cammini in un grafo



```
/* programma che descrive il grafo */

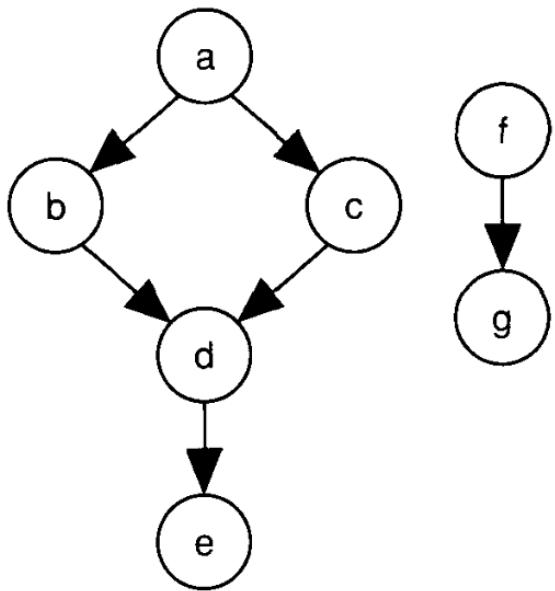
arco(a,b).           arco(b,d).           arco(d,e).
arco(a,c).           arco(c,d).           arco(f,g).
```

Figure 2.4 A simple graph



Regole ricorsive

Cerchiamo ora i cammini in un grafo



```
/* programma che descrive il grafo */

arco(a,b).           arco(b,d).           arco(d,e).
arco(a,c).           arco(c,d).           arco(f,g).
```

Verificare se due nodi sono connessi:

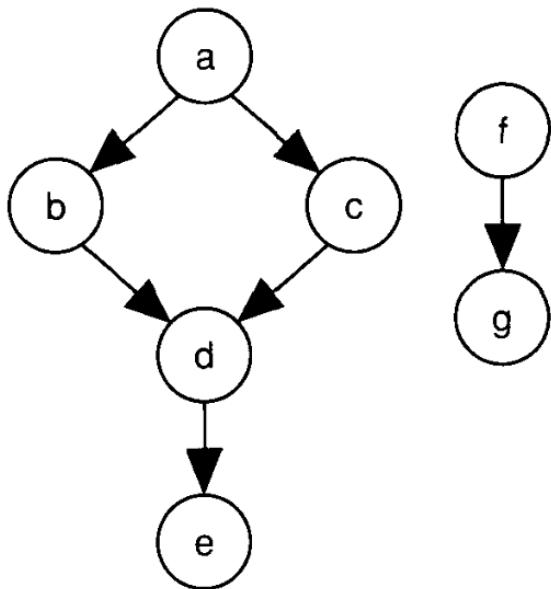
```
connessi(X,X).
```

Figure 2.4 A simple graph



Regole ricorsive

Cerchiamo ora i cammini in un grafo



```
/* programma che descrive il grafo */

arco(a,b).           arco(b,d).           arco(d,e).
arco(a,c).           arco(c,d).           arco(f,g).
```

Verificare se due nodi sono connessi:

```
connessi(X,X).
connessi(X,Y) :- arco(X,Z), connessi(Z,Y).
```

Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`

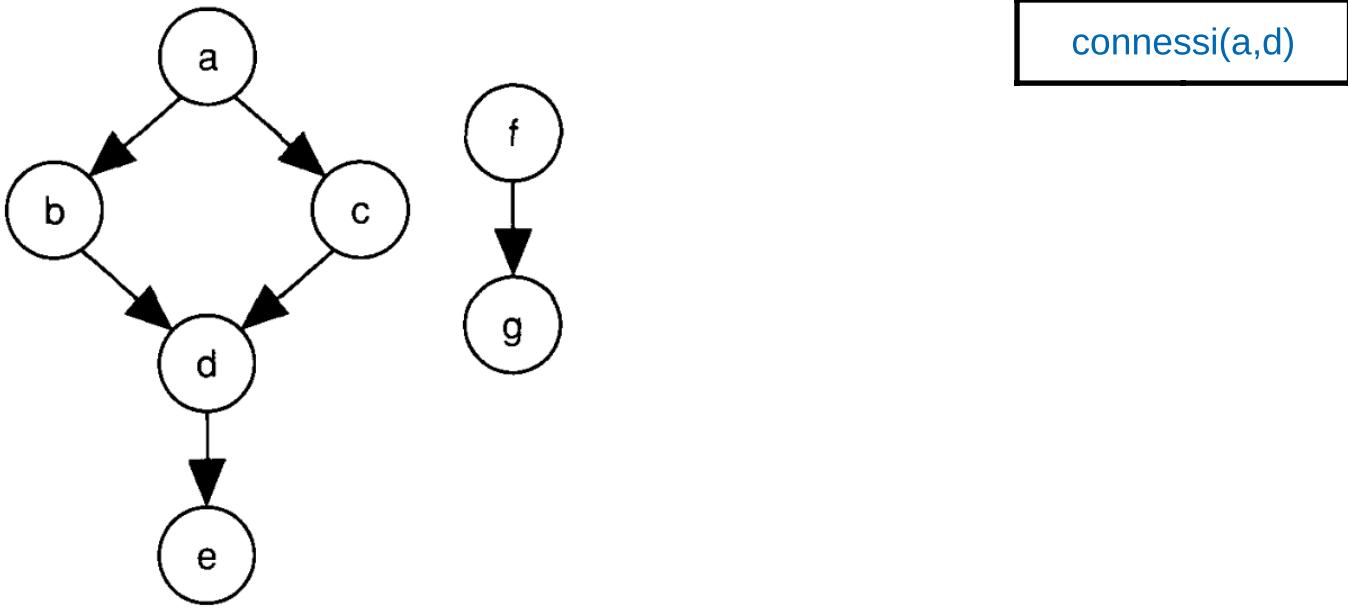


Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`

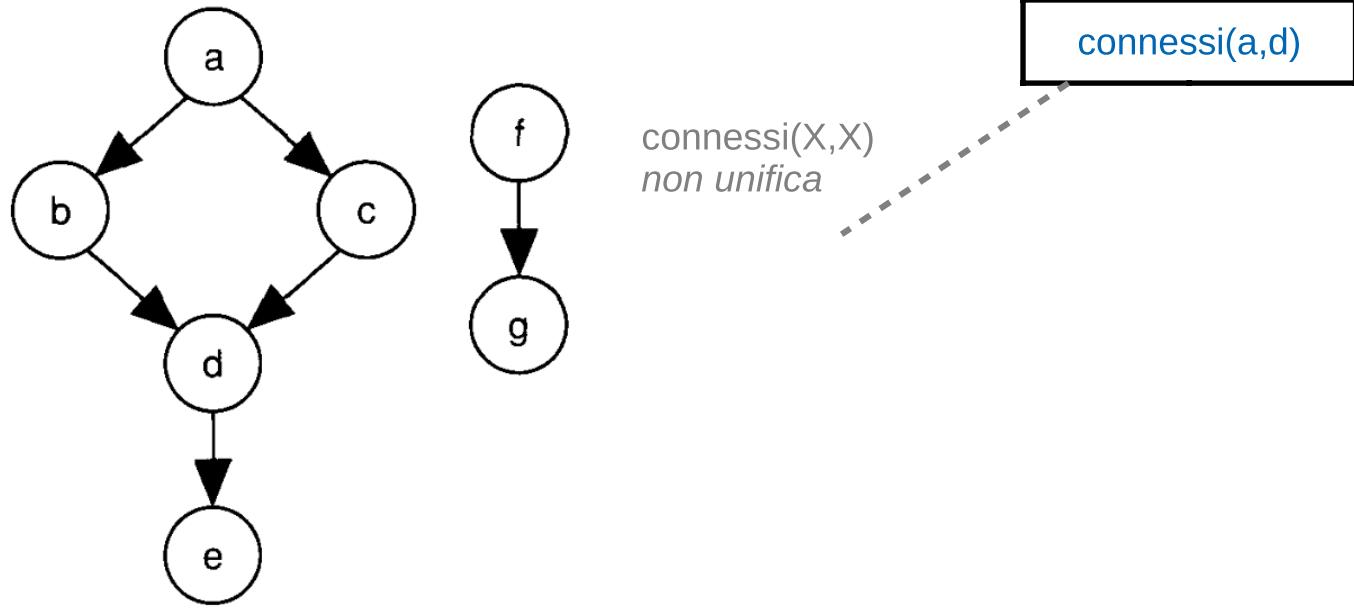
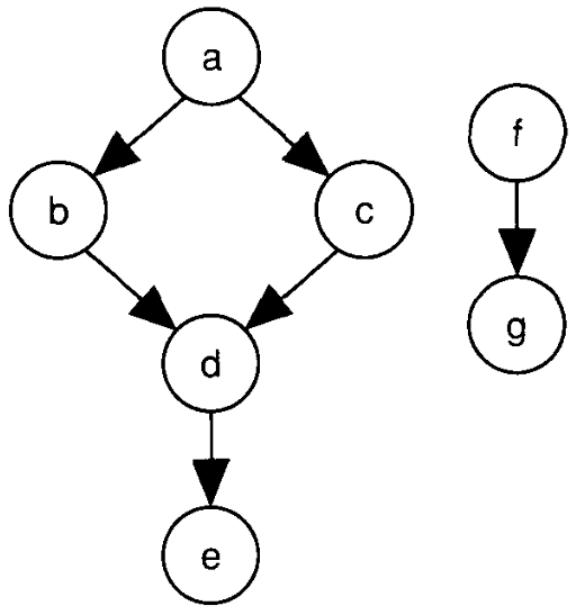


Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`
non unifica

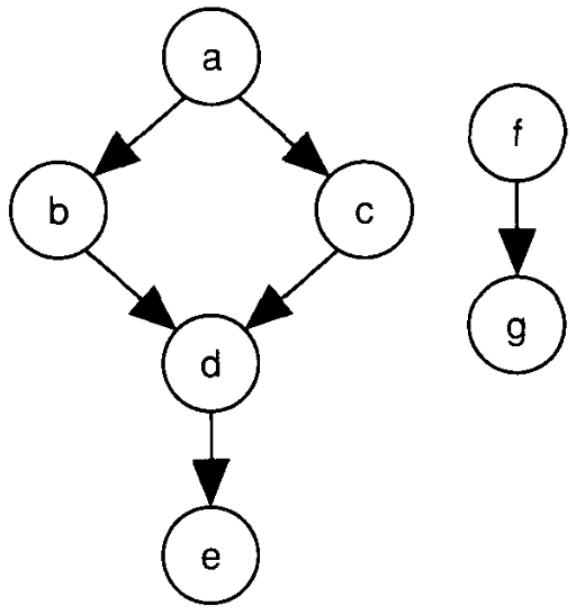
`connessi(a,d)`

`connessi(a,d) :- arco(a,Z1), connessi(Z1,d).`
 $\{ X_1=a , Y_1=d \}$

Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`
non unifica

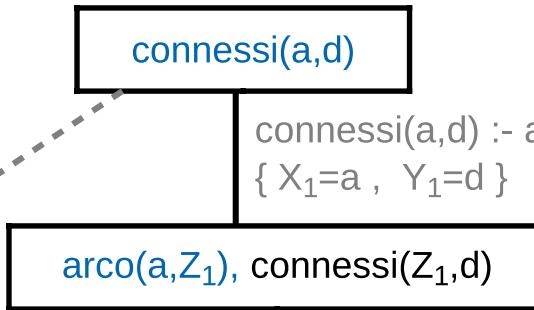
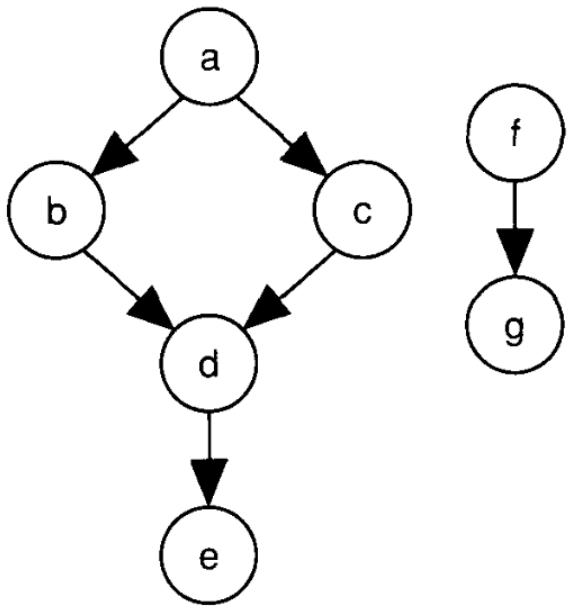


Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`
non unifica

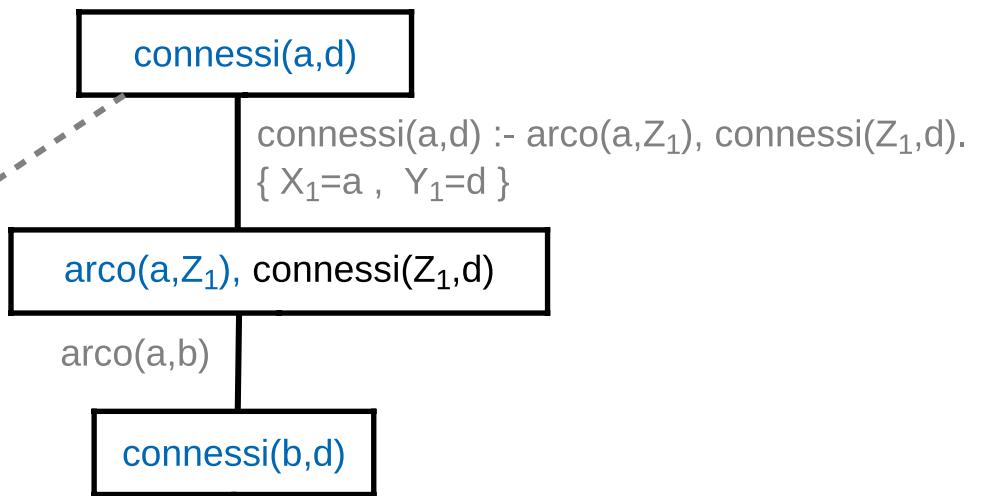
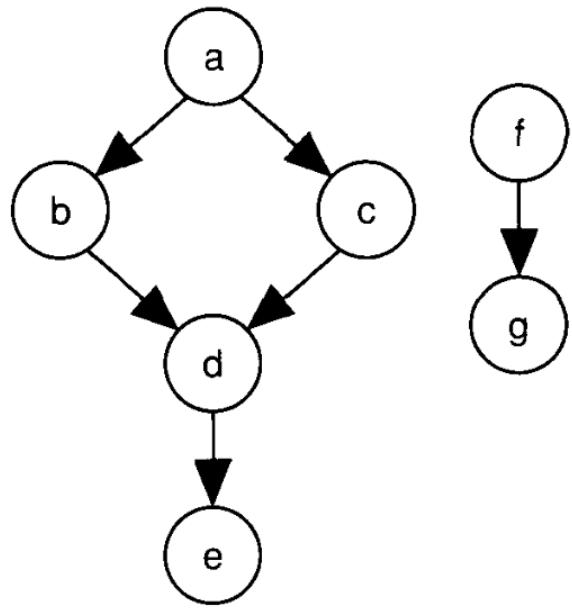


Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`
non unifica

`connessi(X,X)`
non unifica

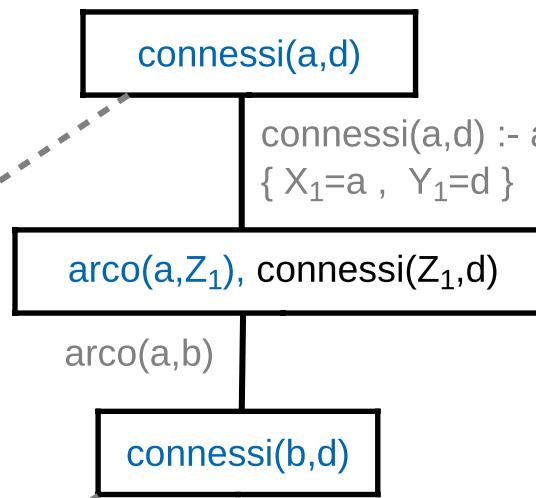
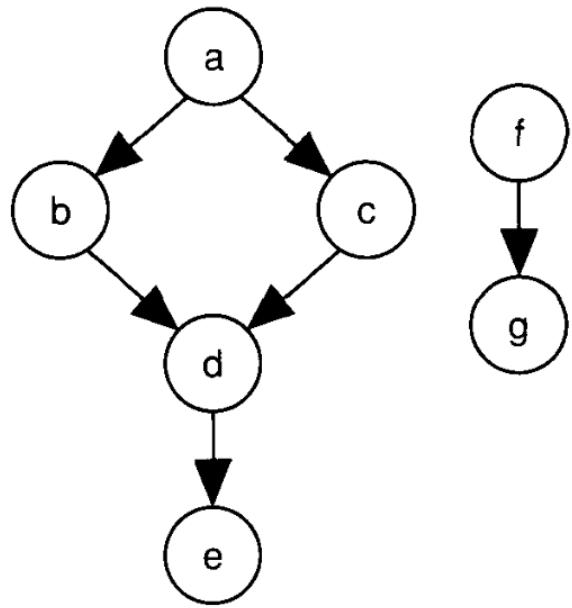


Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`



`connessi(X,X)`
non unifica

`connessi(X,X)`
non unifica

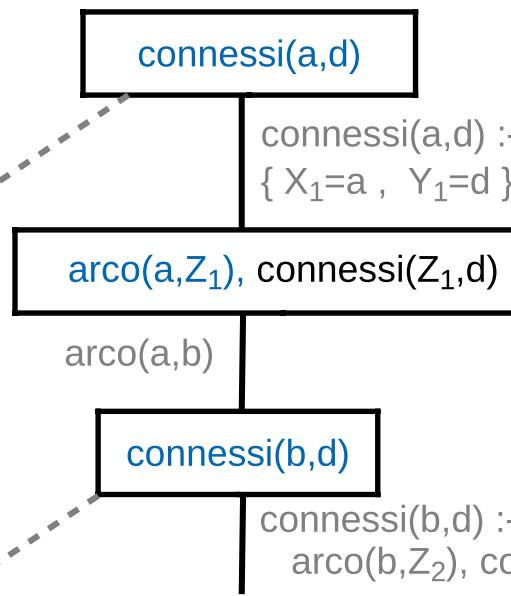


Figure 2.4 A simple graph



Search tree parziale per la query `connessi(a,d)`

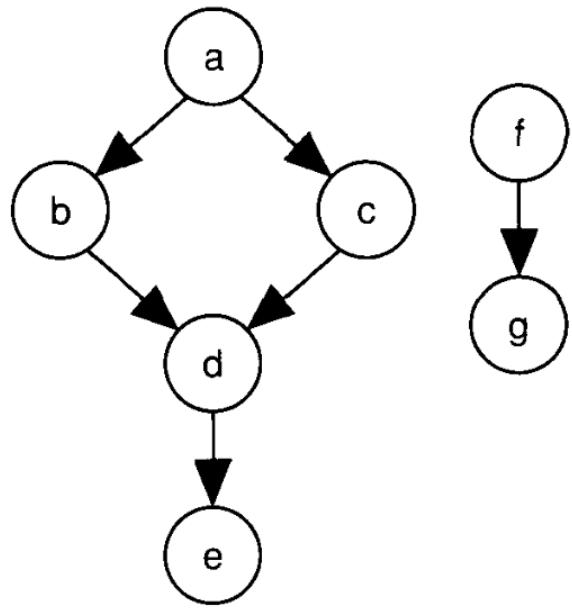
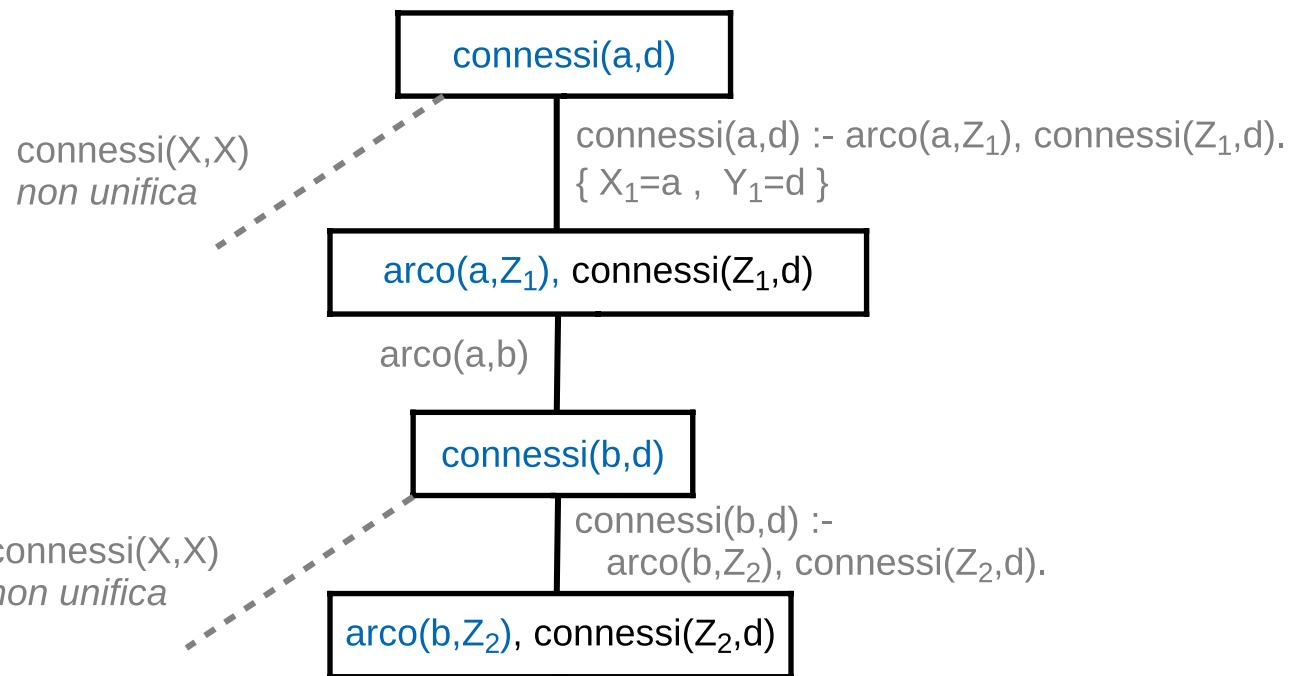


Figure 2.4 A simple graph





Search tree parziale per la query `connessi(a,d)`

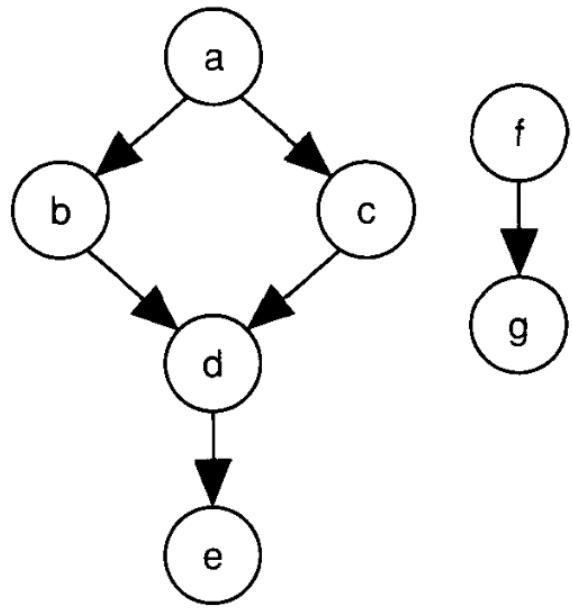
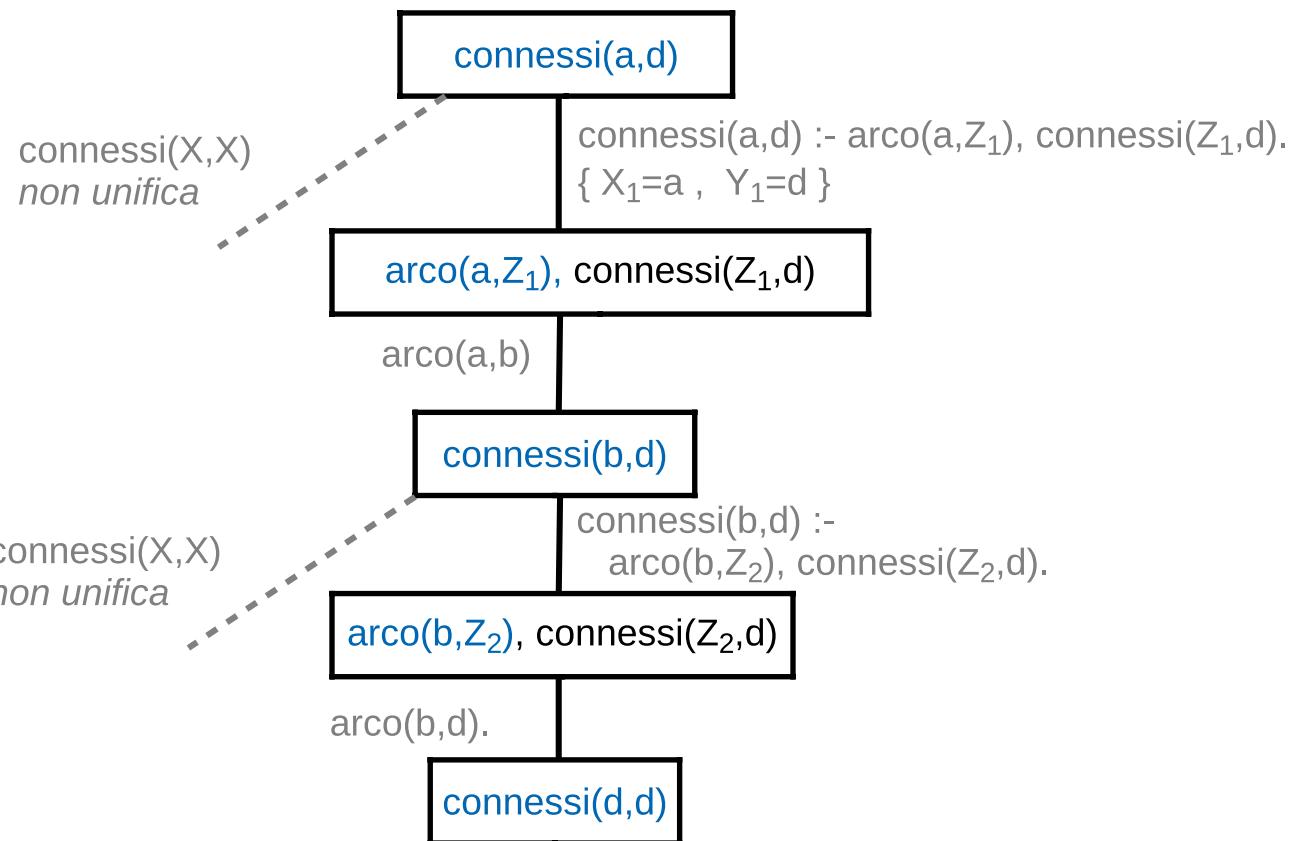


Figure 2.4 A simple graph





Search tree parziale per la query `connessi(a,d)`

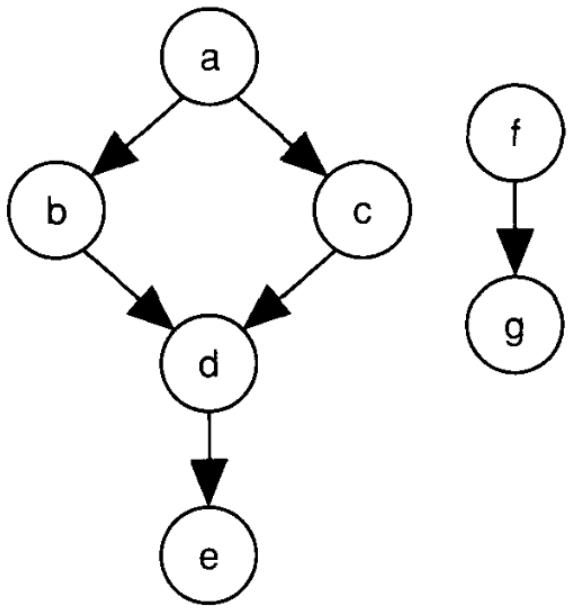
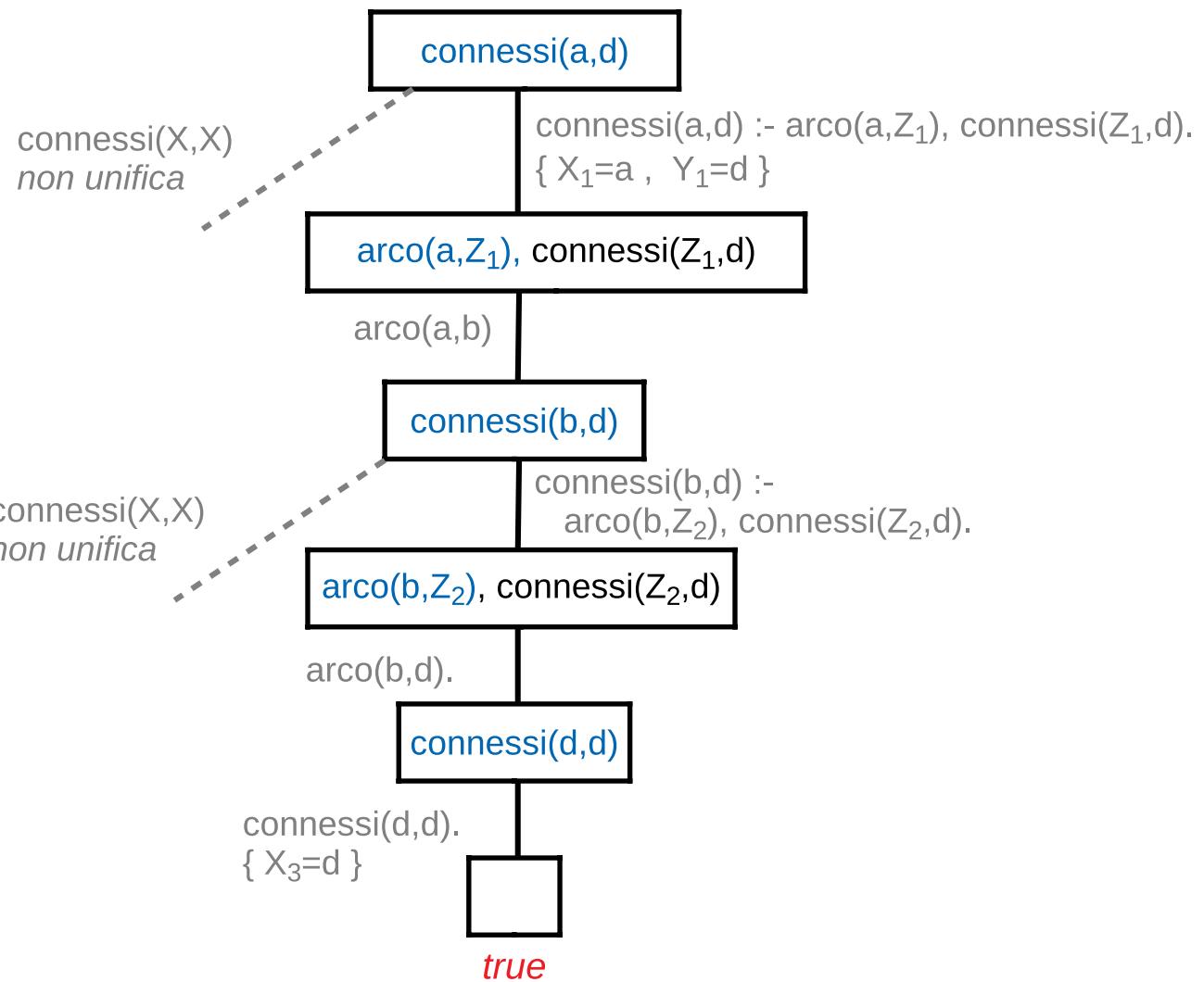


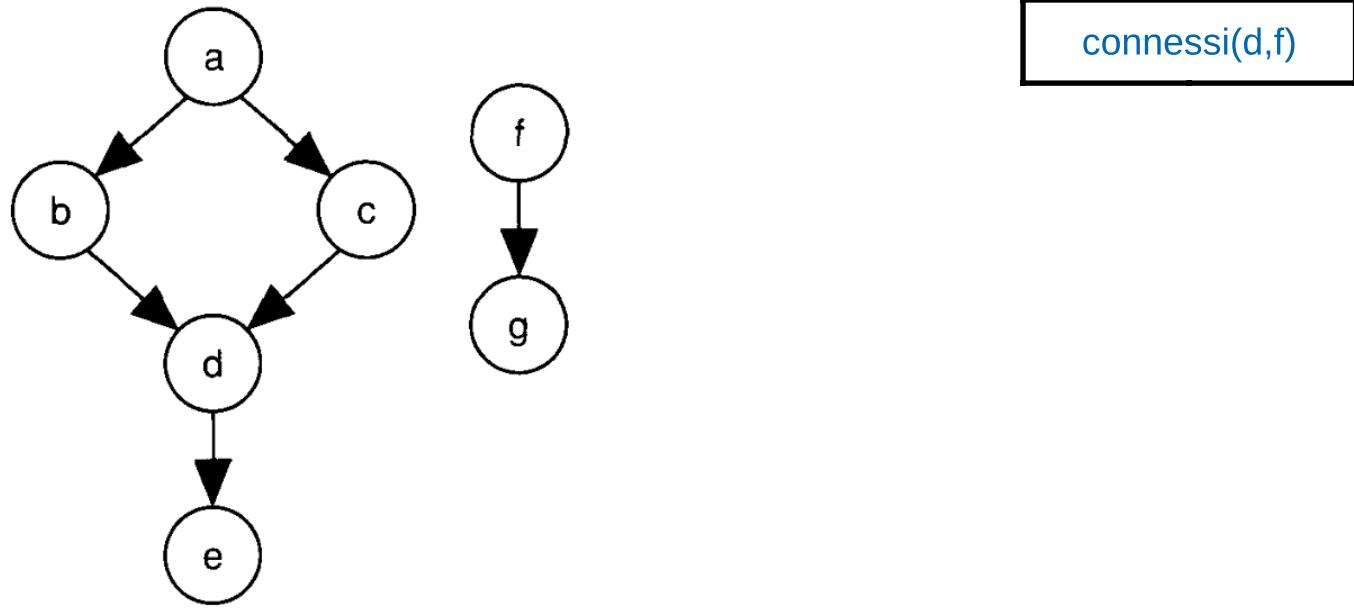
Figure 2.4 A simple graph



Non cerca altre soluzioni perchè la query è ground (inutile restituire altri “true”)



Esempio di search tree per la query $\text{connessi}(d,f)$



connessi(d,f)

Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$

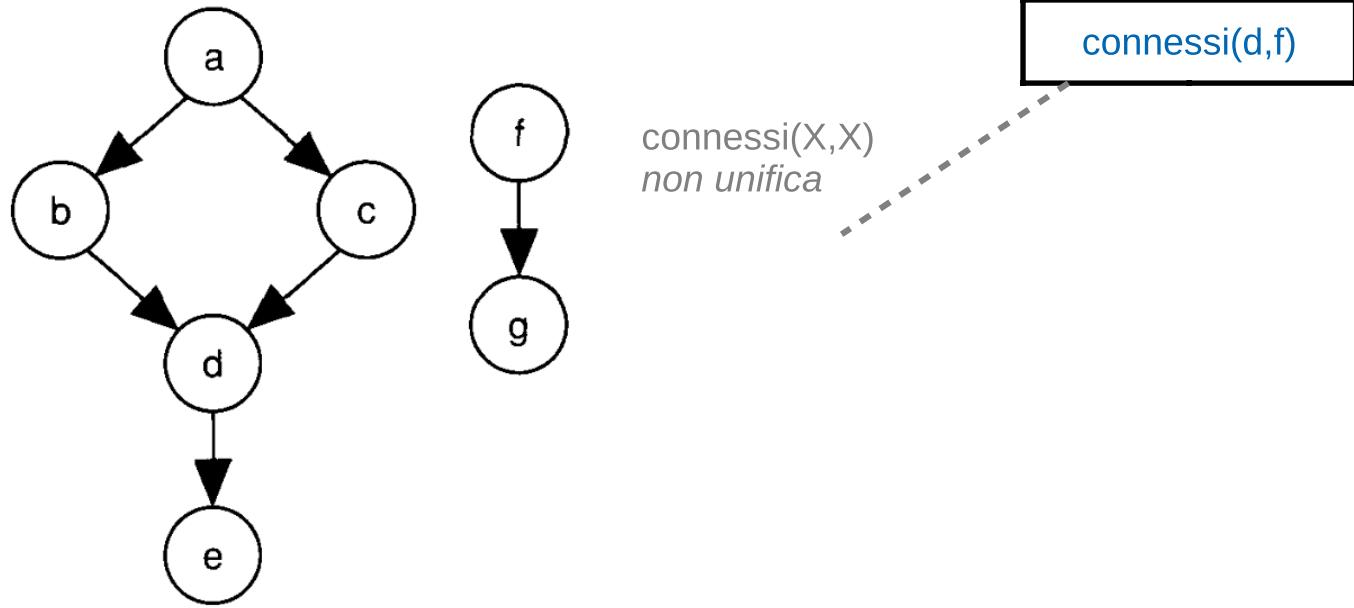
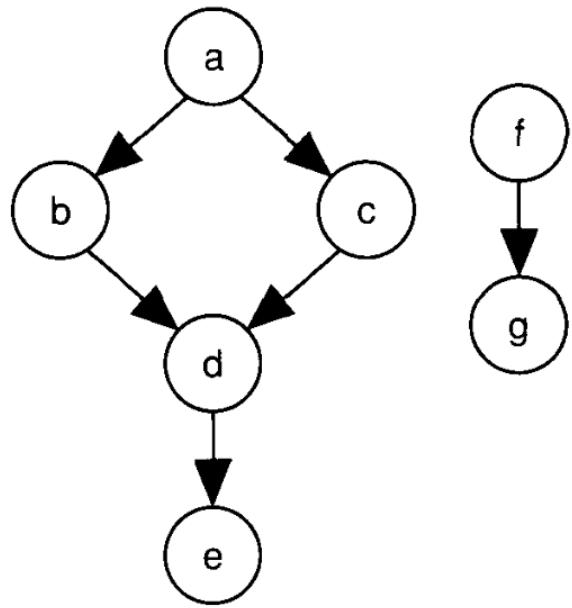


Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$
non unifica

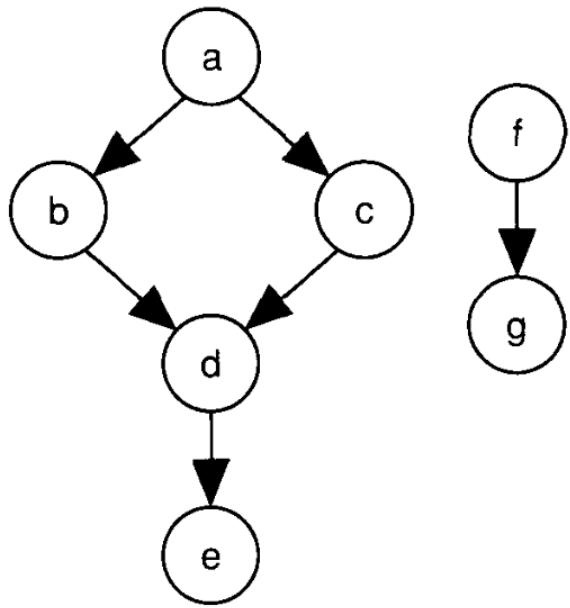
$\text{connessi}(d,f)$

$\text{connessi}(d,f) :- \text{arco}(d,Z_1), \text{connessi}(Z_1,f).$
 $\{ X_1=d , Y_1=f \}$

Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$
non unifica

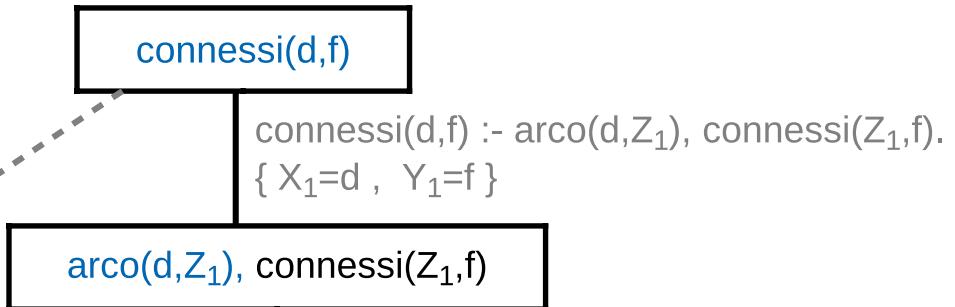
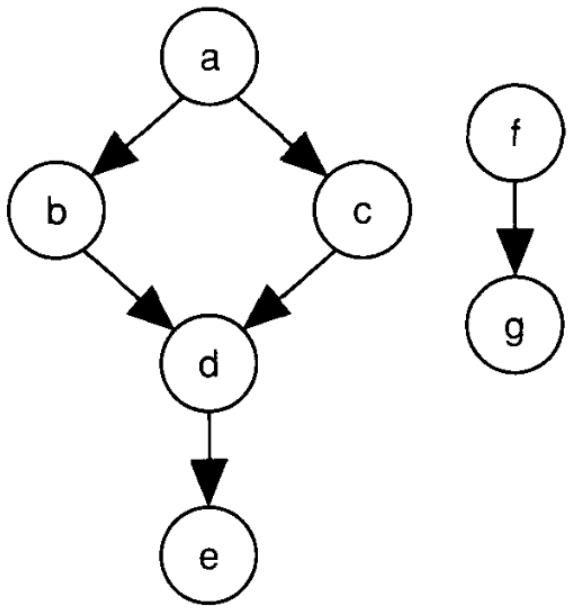


Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$
non unifica

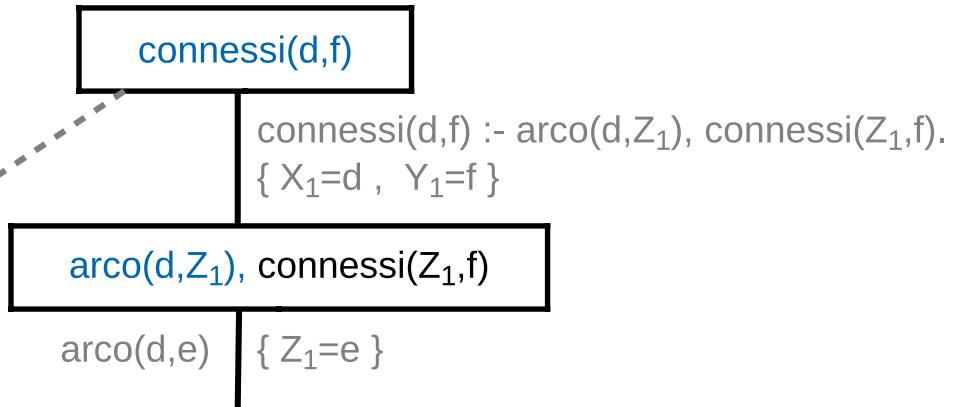
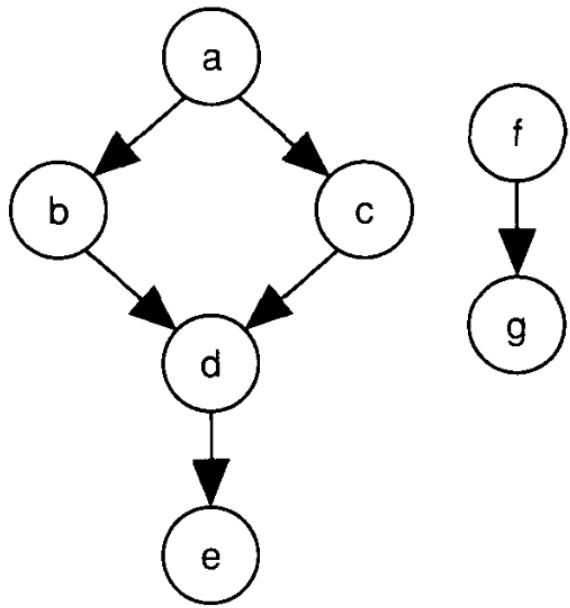


Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$
non unifica

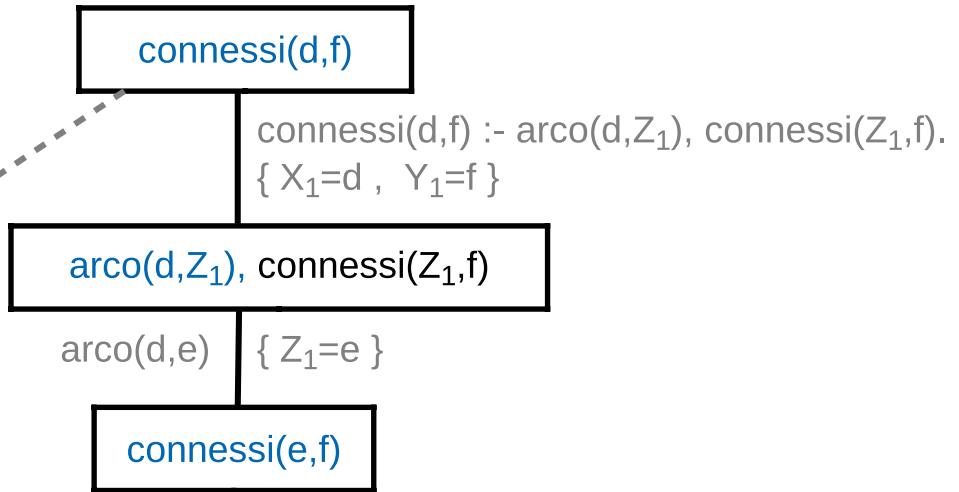
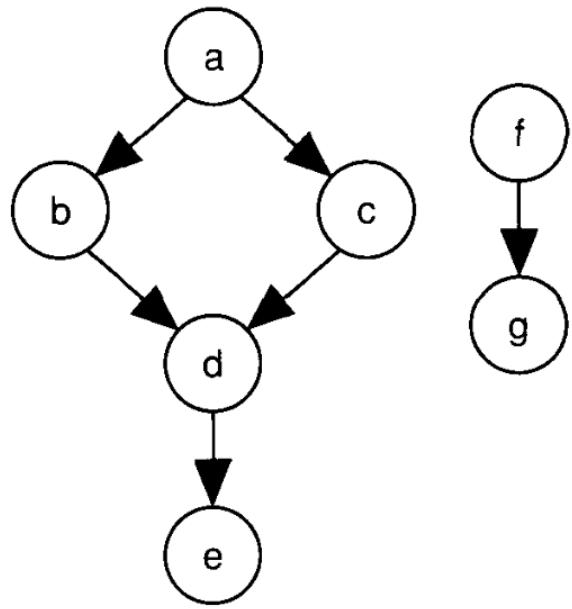


Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$
non unifica

$\text{connessi}(X,X)$
non unifica

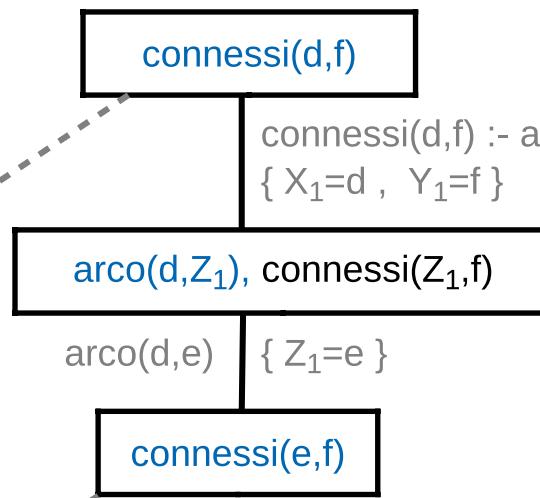
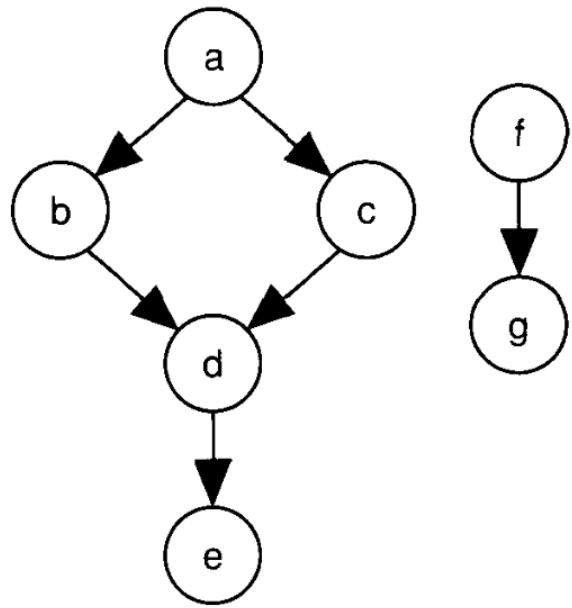


Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$



$\text{connessi}(X,X)$
non unifica

$\text{connessi}(X,X)$
non unifica

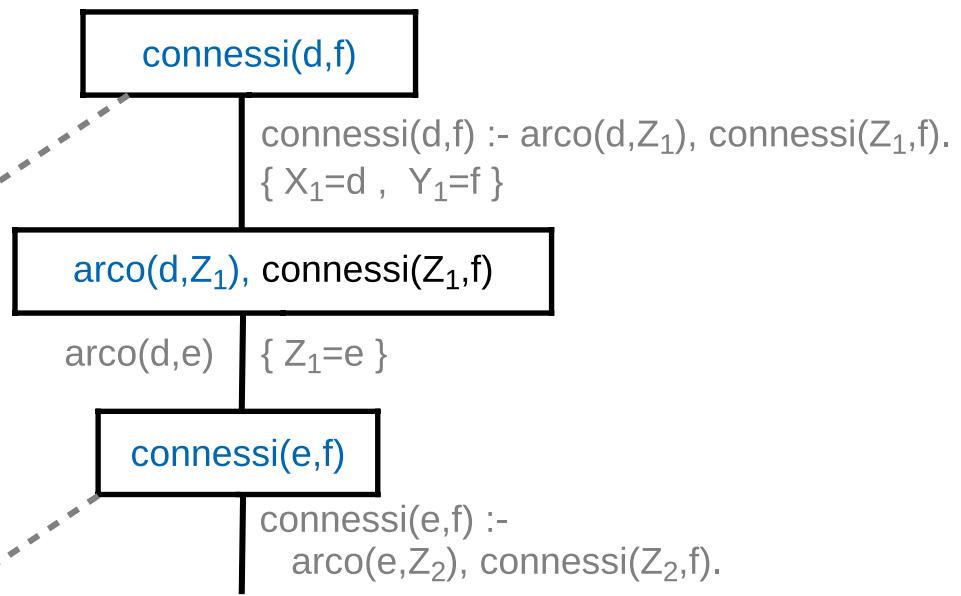


Figure 2.4 A simple graph



Esempio di search tree per la query $\text{connessi}(d,f)$

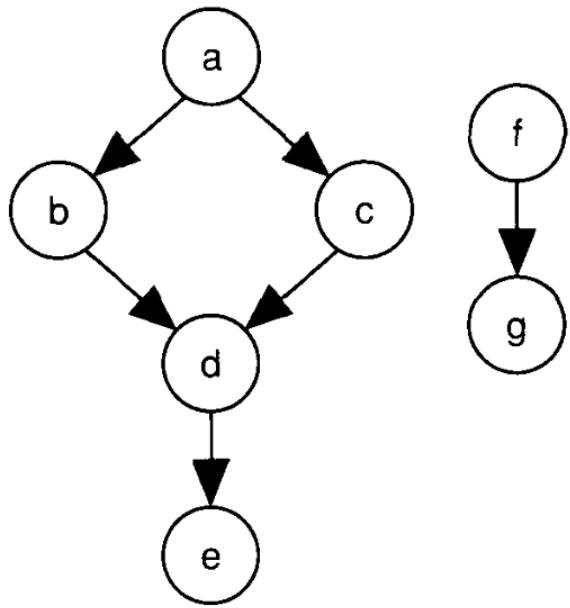
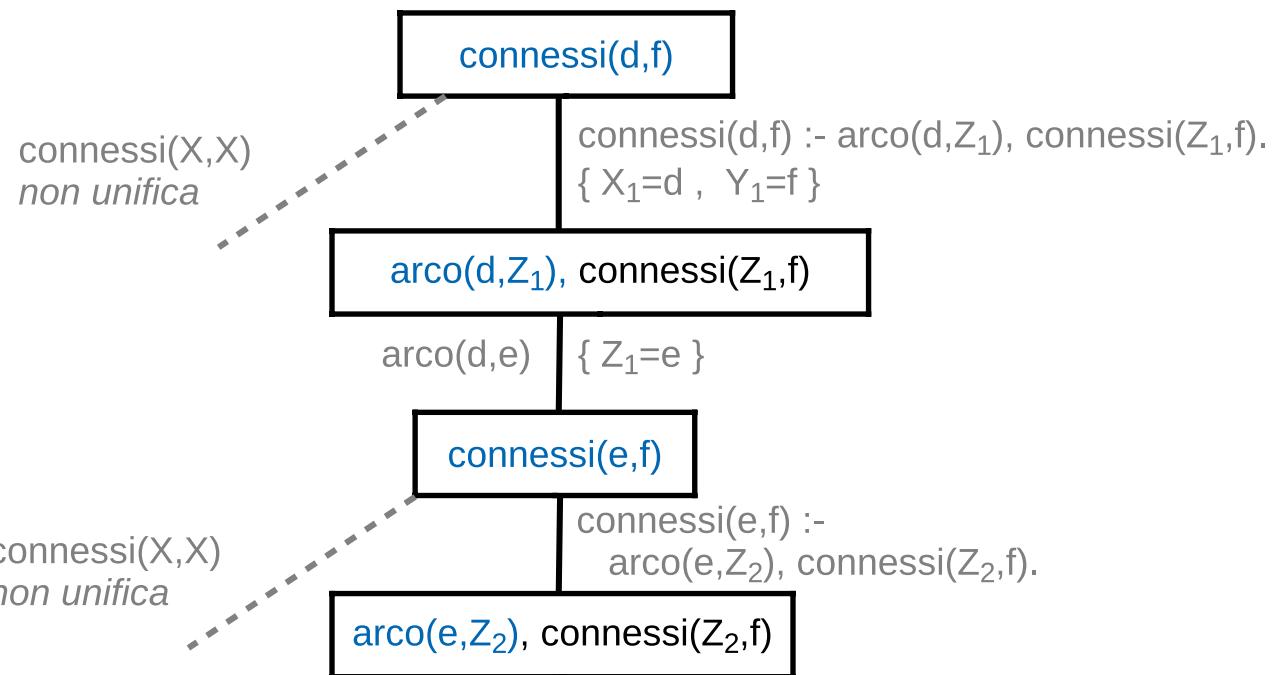


Figure 2.4 A simple graph





Esempio di search tree per la query $\text{connessi}(d,f)$

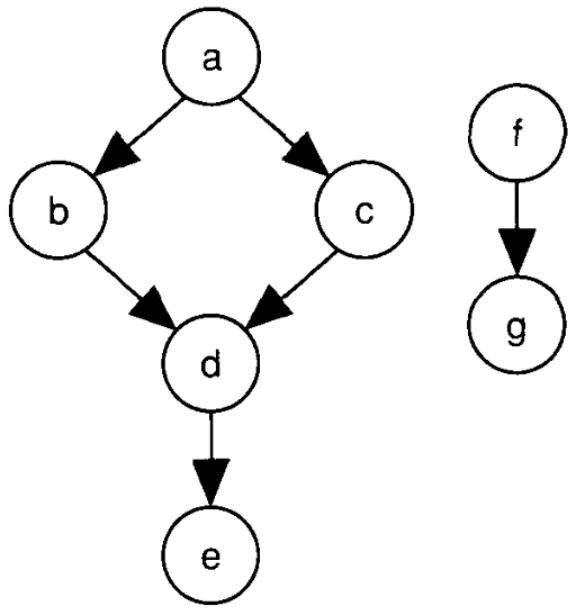
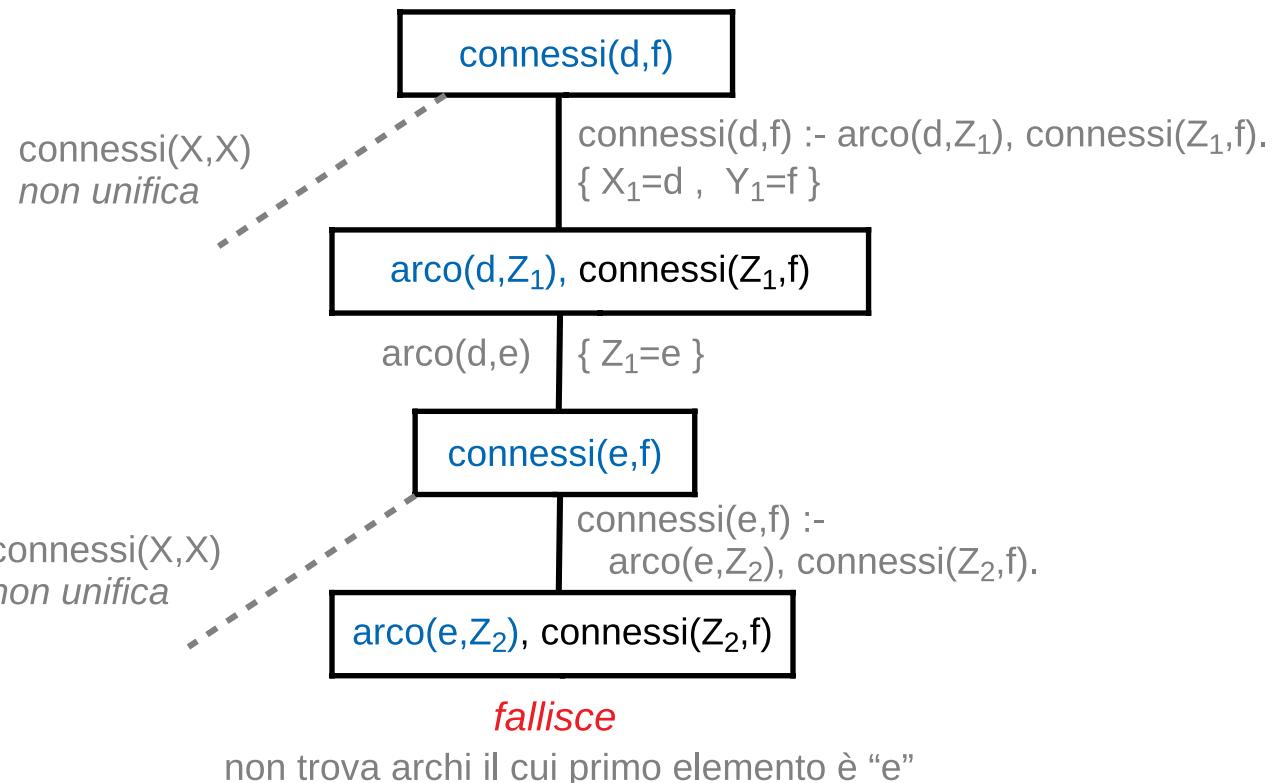


Figure 2.4 A simple graph





Esempio di search tree per la query $\text{connessi}(d,f)$

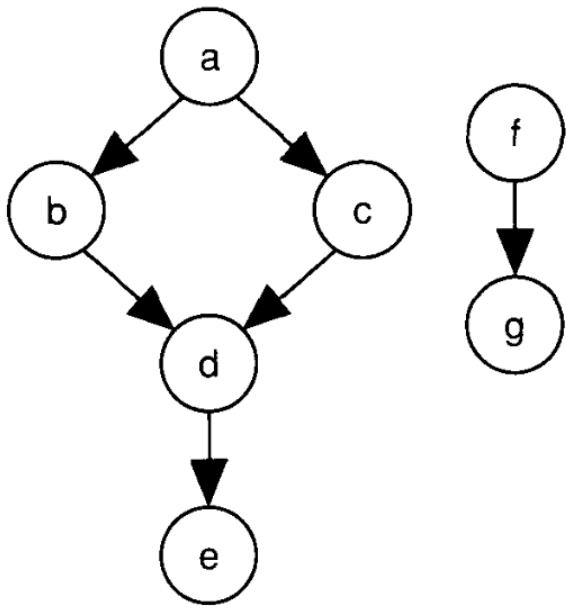
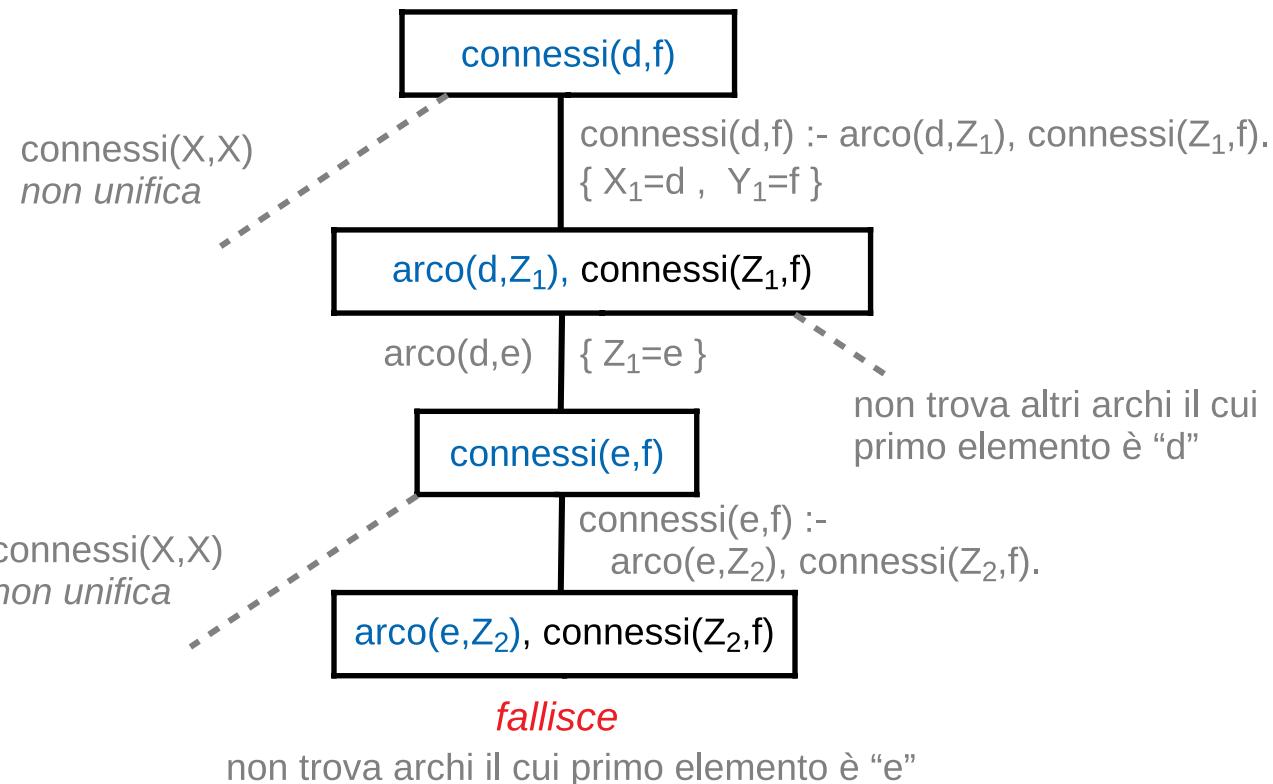


Figure 2.4 A simple graph





Esempio di search tree per la query $\text{connessi}(d,f)$

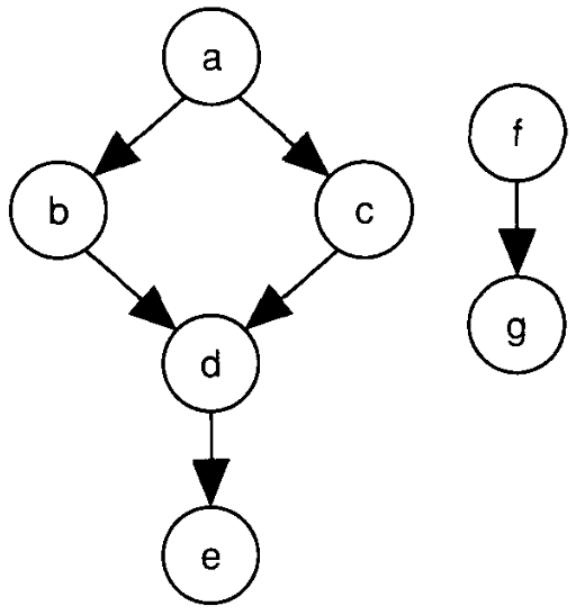
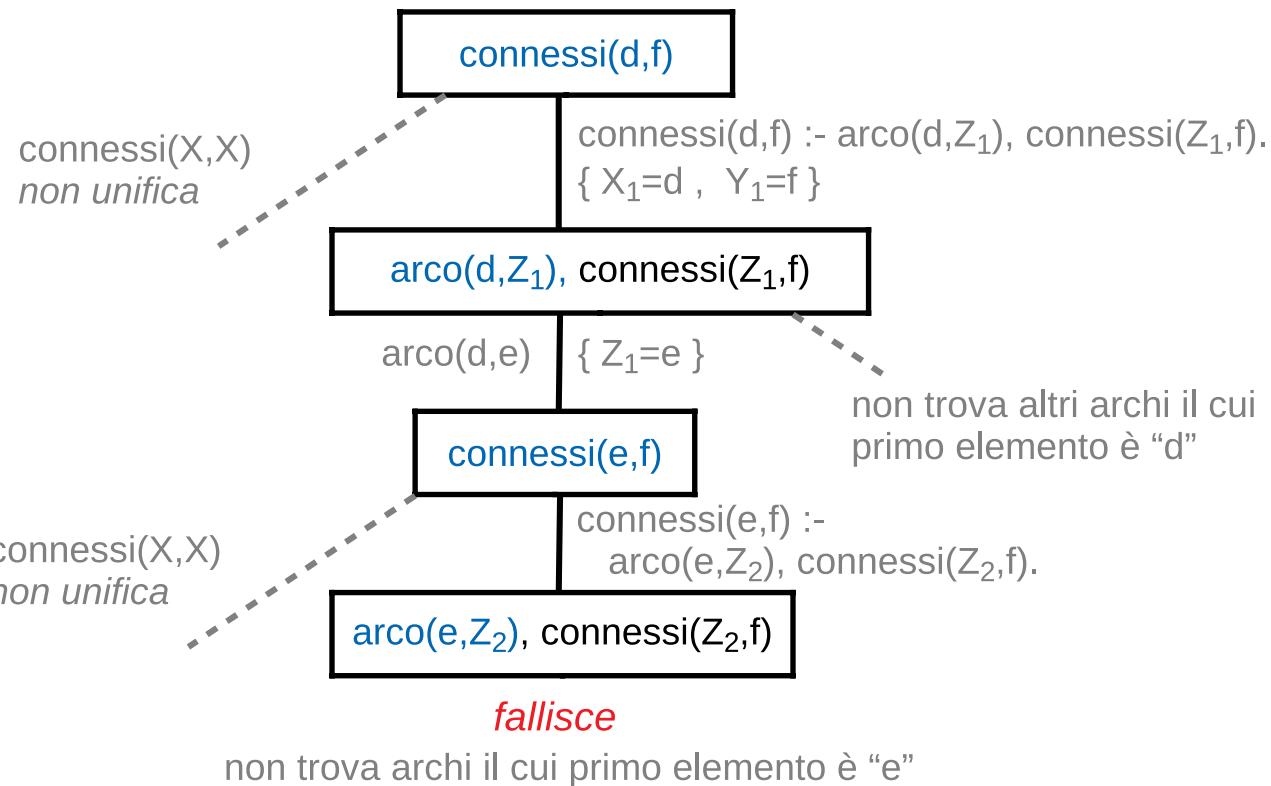


Figure 2.4 A simple graph



La risposta è false



Query non terminanti

Invece delle regole viste prima:

```
connessi(X,X).  
connessi(X,Y) :- arco(X,Z), connessi(Z,Y).
```

...proviamo a usare le seguenti:

```
connessi2(X,X).
```



Query non terminanti

Invece delle regole viste prima:

```
connessi(X,X).  
connessi(X,Y) :- arco(X,Z), connessi(Z,Y).
```

...proviamo a usare le seguenti:

```
connessi2(X,X).  
connessi2(X,Y) :- arco(X,Y).
```



Query non terminanti

Invece delle regole viste prima:

```
connessi(X,X).  
connessi(X,Y) :- arco(X,Z), connessi(Z,Y).
```

...proviamo a usare le seguenti:

```
connessi2(X,X).  
connessi2(X,Y) :- arco(X,Y).  
connessi2(X,Y) :- connessi2(X,Z), connessi2(Z,Y).
```



Query non terminanti

```
connessi2(X,X).  
connessi2(X,Y) :- arco(X,Y).  
connessi2(X,Y) :- connessi2(X,Z), connessi2(Z,Y).
```

Query (vera): `connessi2(a,d)`

Risposta:

```
ERROR: Stack limit (1.0Gb) exceeded  
...  
ERROR: Probable infinite recursion (cycle):  
ERROR: [4,171,061] user:connessi2(a, d)  
ERROR: [4,171,060] user:connessi2(a, d)
```

Esercizio: spiegare l'errore sviluppando il search tree

Conclusione: la definizione di `connessi2` è logicamente corretta, ma non adatta allo schema di valutazione Prolog



Query non terminanti

Attenzione: anche connessi può non terminare, se il grafo contiene cicli!

Ad esempio, aggiungiamo un ciclo al grafo precedente:

```
arco(d,a).
```

Query (falsa): `connessi(a,f)`

Risposta:

```
ERROR: Stack limit (1.0Gb) exceeded  
...
```

Per risolvere questo problema, ci serviranno nuovi costrutti (negazione e liste)



Esempio simile: gli antenati

- Aggiungiamo ora al programma sulla genealogia biblica la nozione di *antenato* (*ancestor*)
- Gli antenati di X sono i suoi genitori, nonni, bisnonni, ...

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Esempio simile: gli antenati

- Aggiungiamo ora al programma sulla genealogia biblica la nozione di *antenato* (*ancestor*)
- Gli antenati di X sono i suoi genitori, nonni, bisnonni, ...
- Definizione ricorsiva: X è un antenato di Y **se** vale una di queste condizioni:
 1. X è genitore di Y (caso base)
 2. X è genitore di Z e Z è antenato di Y (per qualche Z)

```
ancestor(X,Y) :- parent(X,Y).  
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

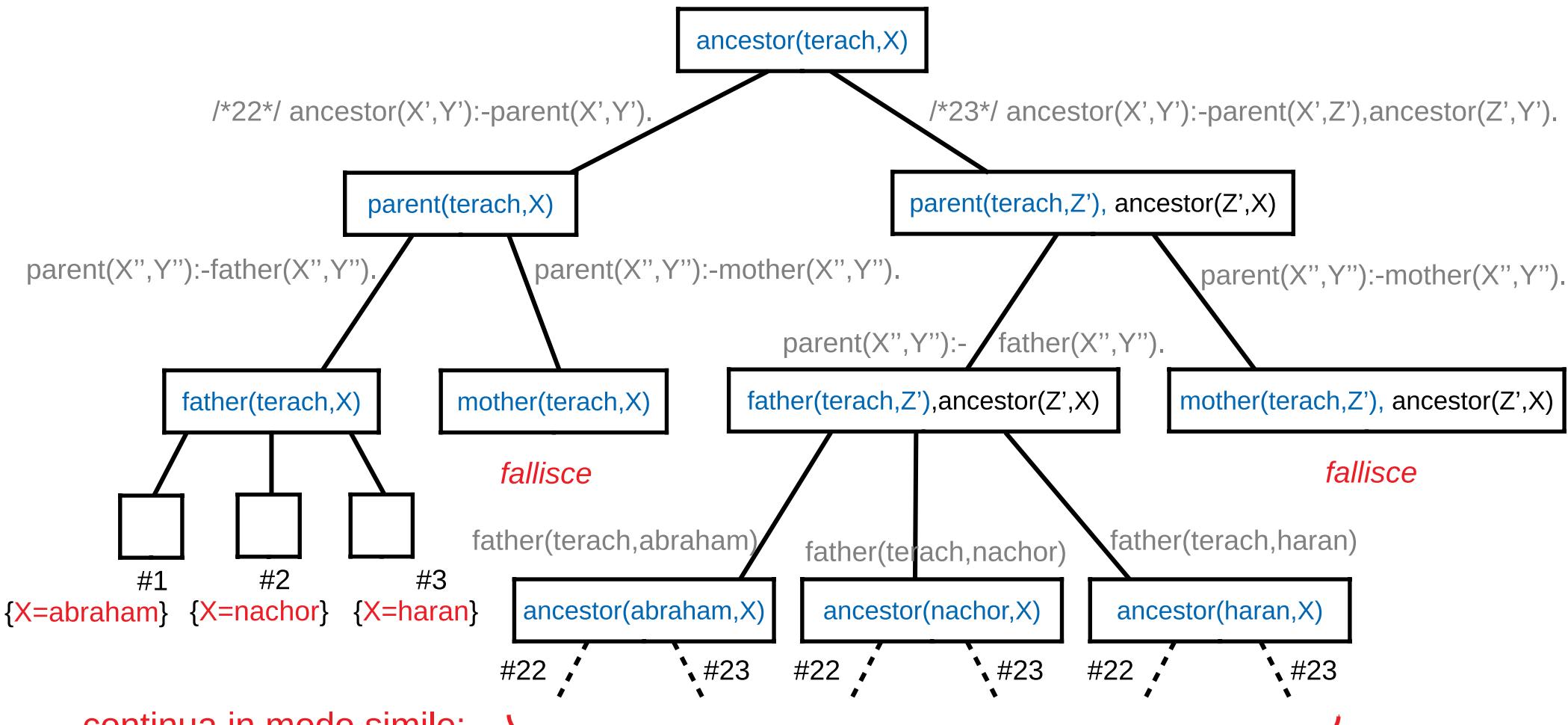
Applicazioni

Programmazione nondeterministica

Unicità di Prolog



Esempio di search tree per ancestor



continua in modo simile:
lega X ai discendenti di
abraham, nachor e haran



Esercizio

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Scrivere un programma Prolog che rappresenta le relazioni di assegnabilità e sottotipo in Java. Il programma deve supportare i seguenti tipi:

- Tutti i tipi primitivi
- I tipi object, string, null
- I tipi person, employee, manager, ciascuno sottotipo del precedente
- I tipi array, tramite un funtore array

Esempi di query:

1. `assegnabile(int, float).` → true
2. `assegnabile(float, int).` → false
3. `sottotipo(int, float).` → false
4. `sottotipo(employee, object).` → true
5. `sottotipo(employee, employee).` → true
6. `sottotipo(array(int), object).` → true
7. `assegnabile(array(employee), array(person)).` → true



Prolog e l'algebra relazionale

- Ogni tabella relazionale r si può rappresentare con dei fatti

$r :$

| | | | |
|-------|-------|-------|-------|
| a_1 | b_1 | c_1 | d_1 |
| a_2 | b_2 | c_2 | d_2 |
| a_3 | b_3 | c_3 | d_3 |
| : | : | : | : |
| : | : | : | : |

```
r(a1, b1, c1, d1).  
r(a2, b2, c2, d2).  
r(a3, b3, c3, d3).  
.  
.  
.
```

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



Prolog e l'algebra relazionale

- Ogni tabella relazionale r si può rappresentare con dei fatti

$r :$

| | | | |
|-------|-------|-------|-------|
| a_1 | b_1 | c_1 | d_1 |
| a_2 | b_2 | c_2 | d_2 |
| a_3 | b_3 | c_3 | d_3 |
| : | : | : | : |
| : | : | : | : |

```
r(a1, b1, c1, d1).  
r(a2, b2, c2, d2).  
r(a3, b3, c3, d3).  
.  
.  
.
```

- Con le regole si può facilmente simulare l'algebra relazionale (ovvero le query SQL)
 - ◆ Prolog genera le n-uple della risposta una per una

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

```
/* UNIONE di r e s */  
r_union_s(X1,...,Xn) :- r(X1,...,Xn).  
r_union_s(X1,...,Xn) :- s(X1,...,Xn).
```



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

[Liste](#)

[Applicazioni](#)

Programmazione
nondeterministica

Unicità di Prolog

```
/* UNIONE di r e s */  
r_union_s(X1,...,Xn) :- r(X1,...,Xn).  
r_union_s(X1,...,Xn) :- s(X1,...,Xn).  
  
/* INTERSEZIONE di r e s */  
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).
```



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

```
/* UNIONE di r e s */  
r_union_s(X1,...,Xn) :- r(X1,...,Xn).  
r_union_s(X1,...,Xn) :- s(X1,...,Xn).  
  
/* INTERSEZIONE di r e s */  
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).  
  
/* PROIEZIONE di r, ad es. su colonne 1 e 3 */  
r13(X1,X3) :- r(X1,...,Xn).
```



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

```
/* UNIONE di r e s */
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).

/* INTERSEZIONE di r e s */
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).

/* PROIEZIONE di r, ad es. su colonne 1 e 3 */
r13(X1,X3) :- r(X1,...,Xn).

/* SELEZIONE di r */
r_sel(X1,...,Xn) :- r(X1,...,Xn), <condizione>.

/* ad esempio: */
r_with_2_less_than_3(X1,...,Xn) :- r(X1,...,Xn), X2 < X3.
```



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

```
/* UNIONE di r e s */
r_union_s(X1,...,Xn) :- r(X1,...,Xn).
r_union_s(X1,...,Xn) :- s(X1,...,Xn).

/* INTERSEZIONE di r e s */
r_inters_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).

/* PROIEZIONE di r, ad es. su colonne 1 e 3 */
r13(X1,X3) :- r(X1,...,Xn).

/* SELEZIONE di r */
r_sel(X1,...,Xn) :- r(X1,...,Xn), <condizione>.

/* ad esempio: */
r_with_2_less_than_3(X1,...,Xn) :- r(X1,...,Xn), X2 < X3.
```

- Scrivere prodotto cartesiano e join su colonne 1 e 2 come esercizio



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da **\+¹**

```
/* DIFFERENZA INSIEMISTICA tra r e s */  
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```



Prolog e l'algebra relazionale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Derivazioni](#)

[Overloading](#)

[Wildcards](#)

[Analisi di circuiti](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Cammini su grafi](#)

[Relational algebra](#)

[Negazione](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da \textbackslash+ ¹

```
/* DIFFERENZA INSIEMISTICA tra r e s */  
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```

- La negazione trasforma false in true, ovvero fallimenti in successi.
 - ◆ $\text{\textbackslash+ p}(X_1, \dots, X_n)$ è *true* se tutti i rami del suo albero di ricerca terminano con un fallimento



Prolog e l'algebra relazionale

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Derivazioni

Overloading

Wildcards

Analisi di circuiti

Cammini su grafi

Cammini su grafi

Cammini su grafi

Cammini su grafi

Relational algebra

Negazione

Liste

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Per la differenza tra relazioni occorre un operatore di **negazione** denotato da \textbackslash+ ¹

```
/* DIFFERENZA INSIEMISTICA tra r e s */  
r_minus_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).
```

- La negazione trasforma false in true, ovvero fallimenti in successi.
 - ◆ $\text{\textbackslash+ p}(X_1, \dots, X_n)$ è *true* se tutti i rami del suo albero di ricerca terminano con un fallimento
 - ◆ L'albero deve essere finito. Se un programma cade in una ricorsione infinita ovviamente non termina.

¹Nel libro si usa **not**



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Liste



Le liste in Prolog

- La rappresentazione è analoga a quella in ML
 - ◆ costruttore lista vuota: []
 - ◆ costruttore nodi: [elem|resto]

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

| Abbreviata | Costruttori espliciti |
|------------|-----------------------|
| [a] | [a []] |



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Sintassi](#)

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]

[a , b]

Costruttori esplicativi

[a | []]

[a | [b | []]]



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]

[a , b]

[a , b , c]

Costruttori esplicativi

[a | []]

[a | [b | []]]

[a | [b | [c | []]]]



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]
[a, b]
[a, b, c]
[a | X]
[a, b | X]

Costruttori espliciti

[a | []]
[a | [b | []]]
[a | [b | [c | []]]]
[a | X]
[a | [b | X]]



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]
[a, b]
[a, b, c]
[a|X]
[a, b|X]
[a, b, c|X]

Costruttori espliciti

[a | []]
[a | [b | []]]
[a | [b | [c | []]]]
[a | X]
[a | [b | X]]
[a | [b | [c | X]]]



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]
[a, b]
[a, b, c]
[a|X]
[a, b|X]
[a, b, c|X]

Costruttori espliciti

[a | []]
[a | [b | []]]
[a | [b | [c | []]]]
[a | X]
[a | [b | X]]
[a | [b | [c | X]]]



Le liste in Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

■ La rappresentazione è analoga a quella in ML

- ◆ costruttore lista vuota: []
- ◆ costruttore nodi: [elem|resto]

■ Notazioni alternative equivalenti:

Abbreviata

[a]
[a, b]
[a, b, c]
[a | X]
[a, b | X]
[a, b, c | X]

Costruttori espliciti

[a | []]
[a | [b | []]]
[a | [b | [c | []]]]
[a | X]
[a | [b | X]]
[a | [b | [c | X]]]

■ Notare la possibilità di esprimere liste *parzialmente specificate*, dove alcuni elementi ed eventualmente la coda sono variabili.

[a , **X** , b | **Y**]



Il predicato member

■ Il predicato member cerca un elemento X in una lista L

- ◆ è ovviamente *ricorsivo*
- ◆ usiamo la wildcard ‘_’

```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Sintassi](#)

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Il predicato member

- Il predicato member cerca un elemento X in una lista L

- ◆ è ovviamente *ricorsivo*
 - ◆ usiamo la wildcard ‘_’

```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```

- Somiglia alla definizione per casi in ML

- ◆ ma in Prolog posso usare la stessa variabile più volte
 - ◆ per esprimere pattern dove certi elementi sono uguali

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



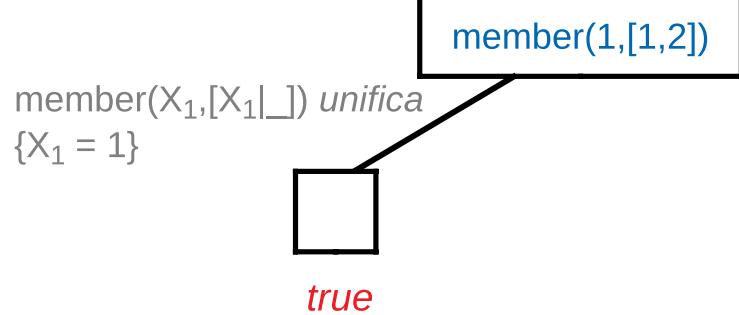
Esempi di derivazione per member

member(1,[1,2])

```
member(X, [X | _]).  
member(X, [_ | L]) :- member(X, L).
```



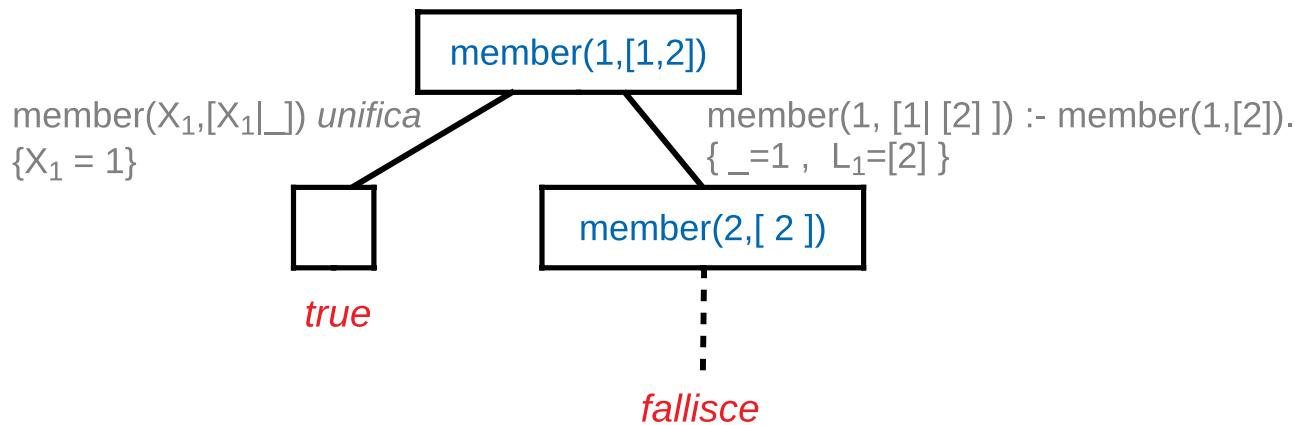
Esempi di derivazione per member



```
member(X, [X | _]).  
member(X, [_ | L]) :- member(X, L).
```



Esempi di derivazione per member



Questo ramo non viene esplorato perché la query è ground

```
member(X, [X|_]).  
member(X, [_|L]) :- member(X,L).
```



Esempi di derivazione per member

member(2,[1,2])

```
member(X, [X | _]).  
member(X, [_ | L]) :- member(X, L).
```



Esempi di derivazione per member

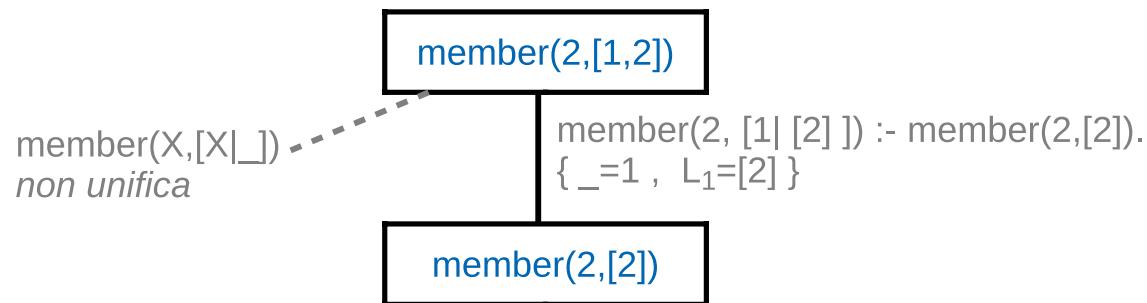
member(2,[1,2])

member(X,[X|_])
non unifica

```
member(X, [X | _]).  
member(X, [_ | L]) :- member(X, L).
```



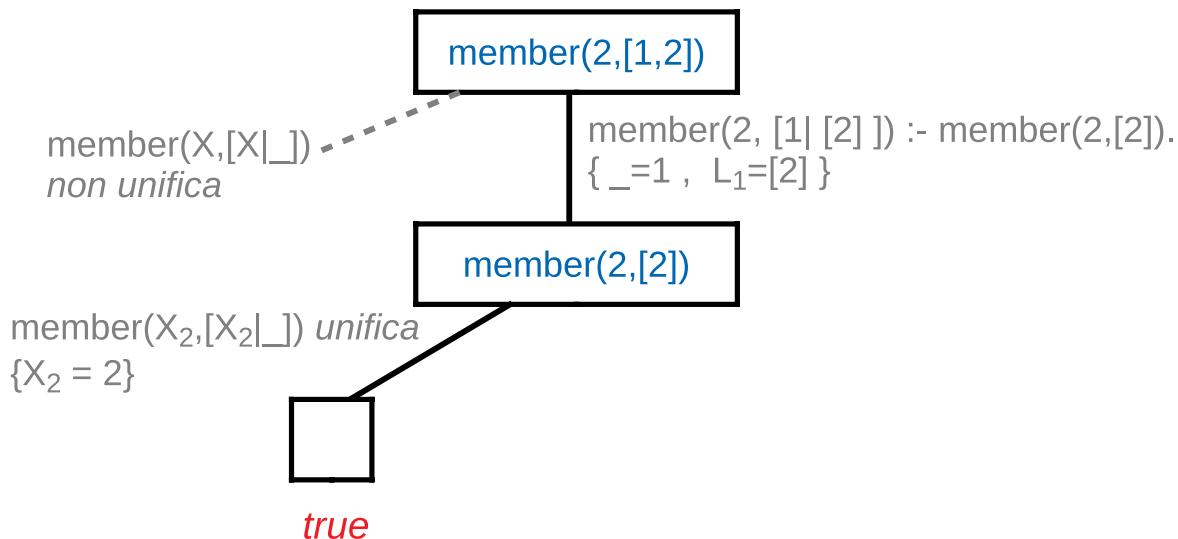
Esempi di derivazione per member



```
member(X, [X|_]).  
member(X, [_|L]) :- member(X,L).
```



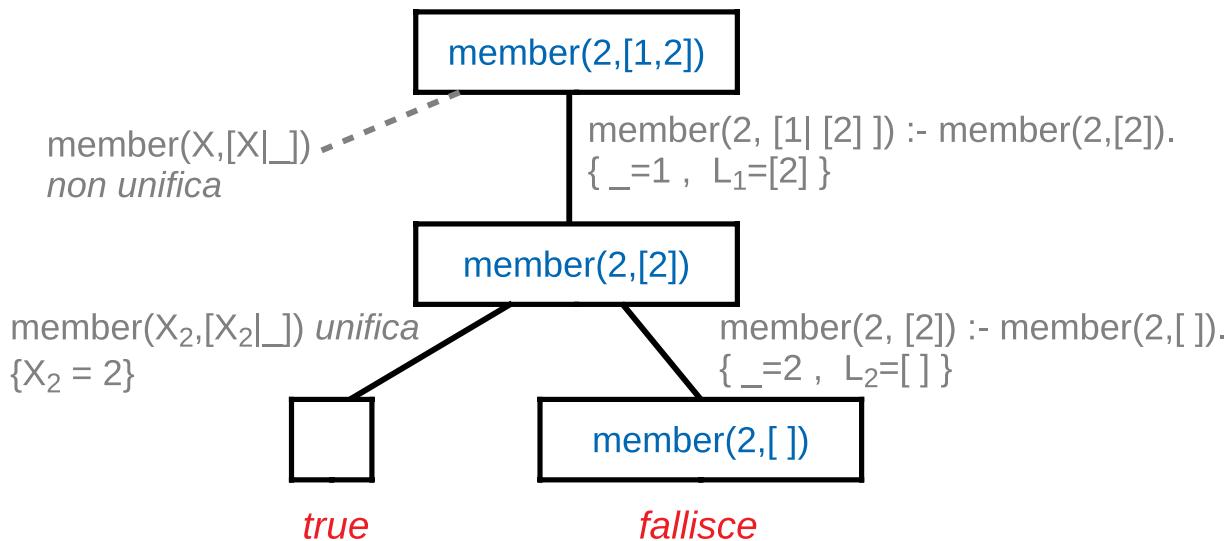
Esempi di derivazione per member



```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```



Esempi di derivazione per member

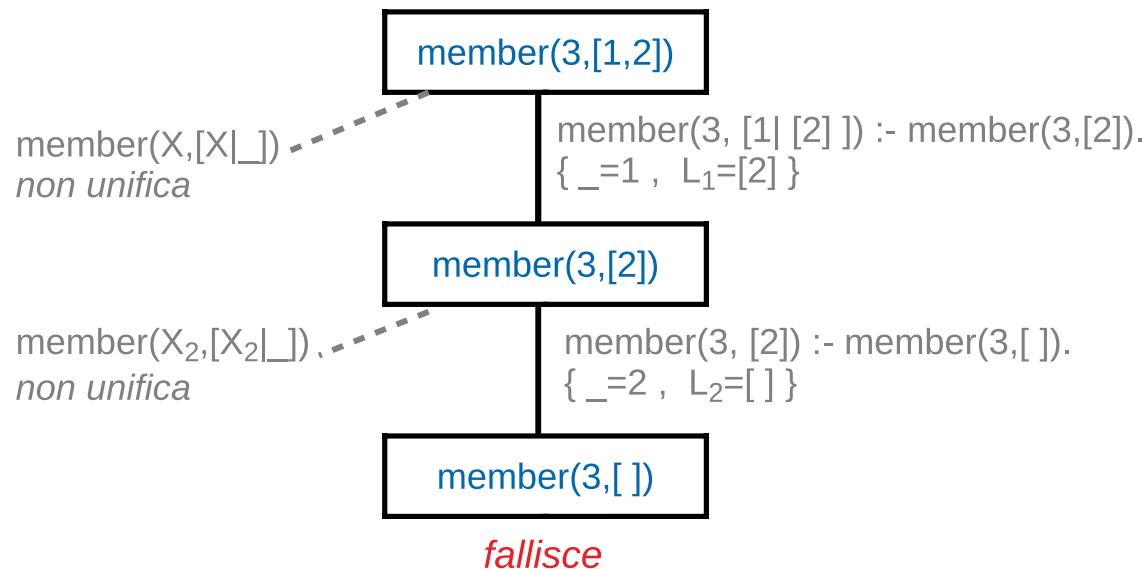


Questo ramo non viene esplorato perché la query è ground

```
member(X, [X|_]).  
member(X, [_|L]) :- member(X,L).
```



Esempi di derivazione per member

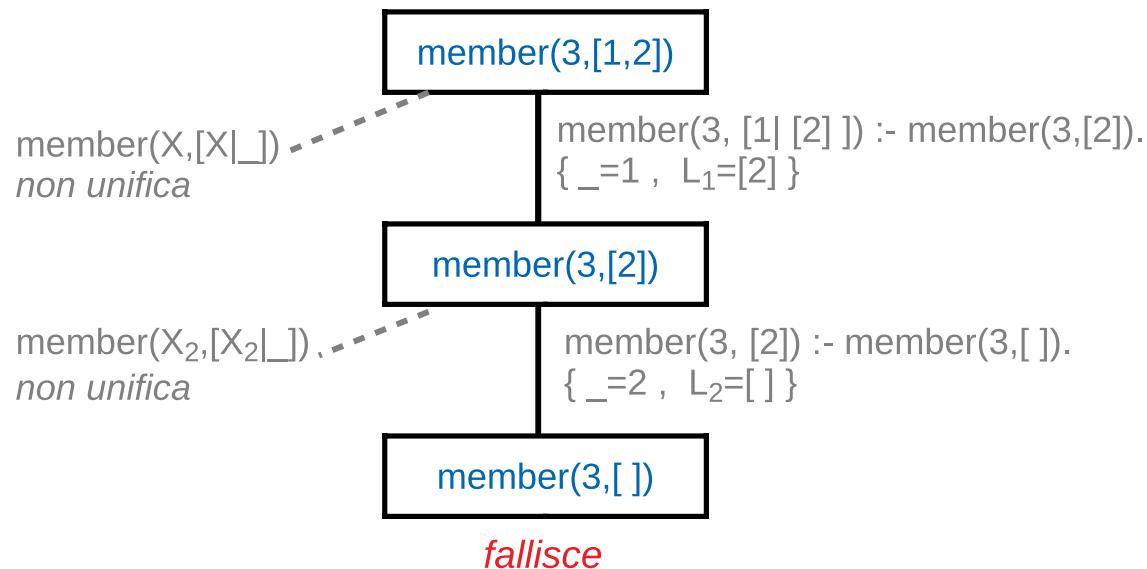


Non ci sono altri rami da provare
→ Risponde *false*

```
member(X,[X|_]).  
member(X,[_|L]) :- member(X,L).
```



Esempi di derivazione per member



Non ci sono altri rami da provare
→ Risponde *false*

```
member(X,[X|_]).  
member(X,[_|L]) :- member(X,L).
```

- Notare che in ML avrebbe sollevato una eccezione. Qui restituisce *false*



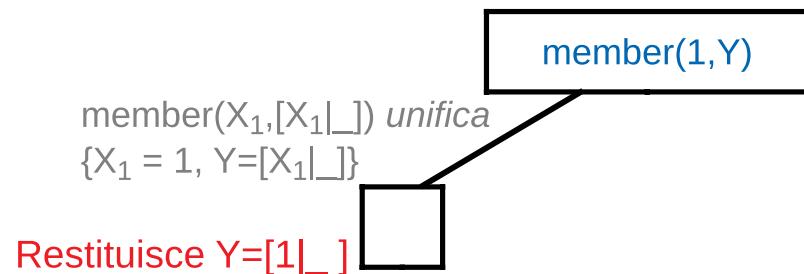
Esempi di derivazione per member

member(1,Y)

```
member(X,[X|_]).  
member(X,[_|L]) :- member(X,L).
```



Esempi di derivazione per member



```
member(X,[X|_]).  
member(X,[_|L]) :- member(X,L).
```

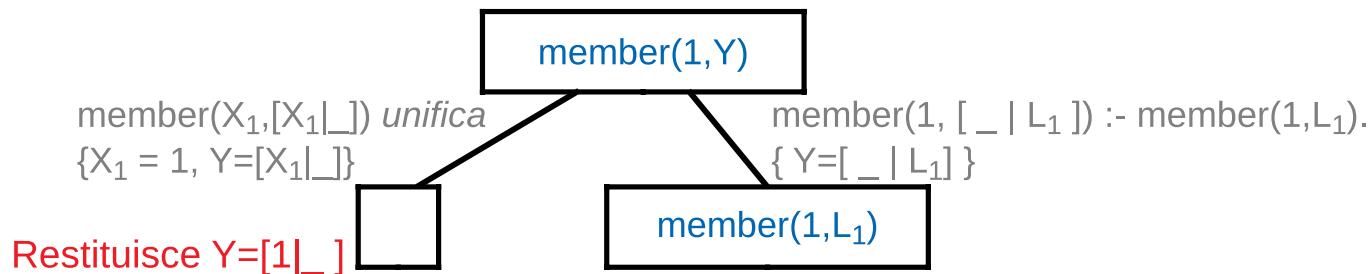
Il valore di Y si ottiene così:

X₁=1,

Y=[X₁|_] = [1|_]



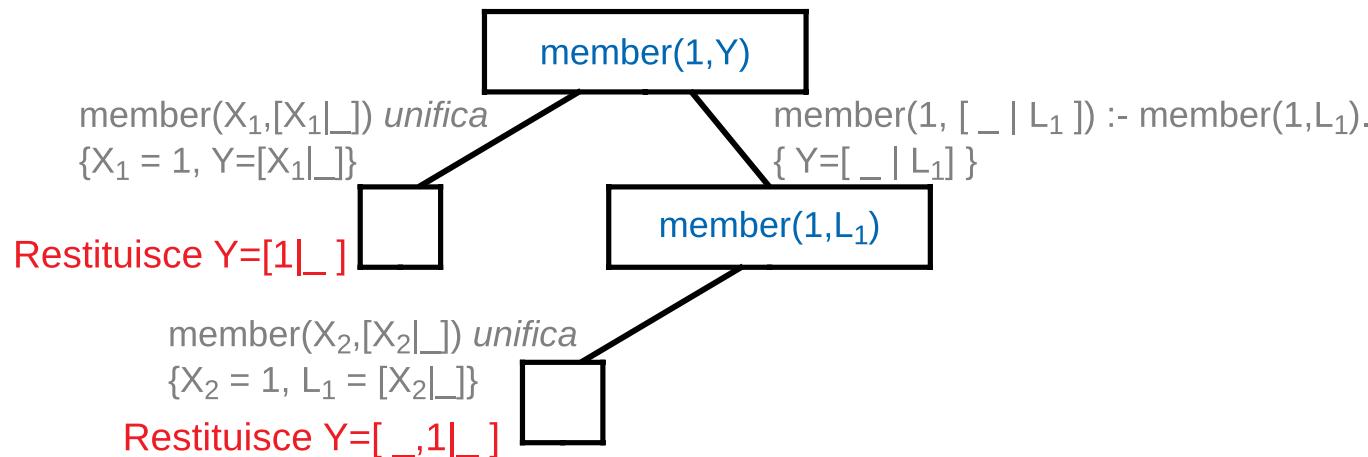
Esempi di derivazione per member



```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```



Esempi di derivazione per member



```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```

Il valore di Y si ottiene così:

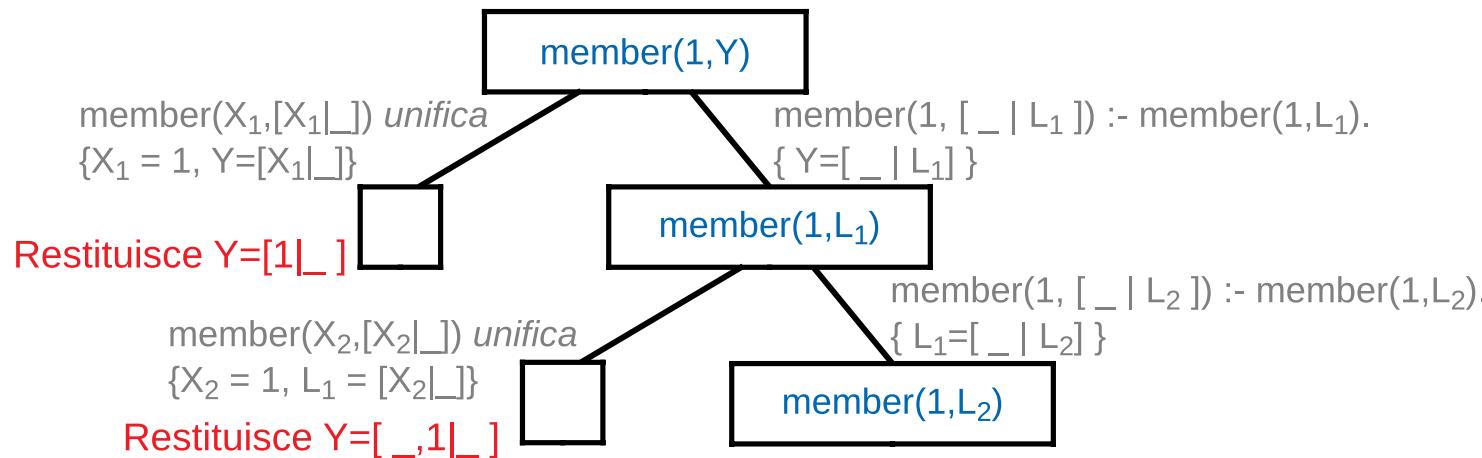
$$X_2 = 1,$$

$$L_1 = [X_2|_] = [1|_],$$

$$Y = [_ | L_1] = [_ | [1|_]] = [_ , 1 | _]$$



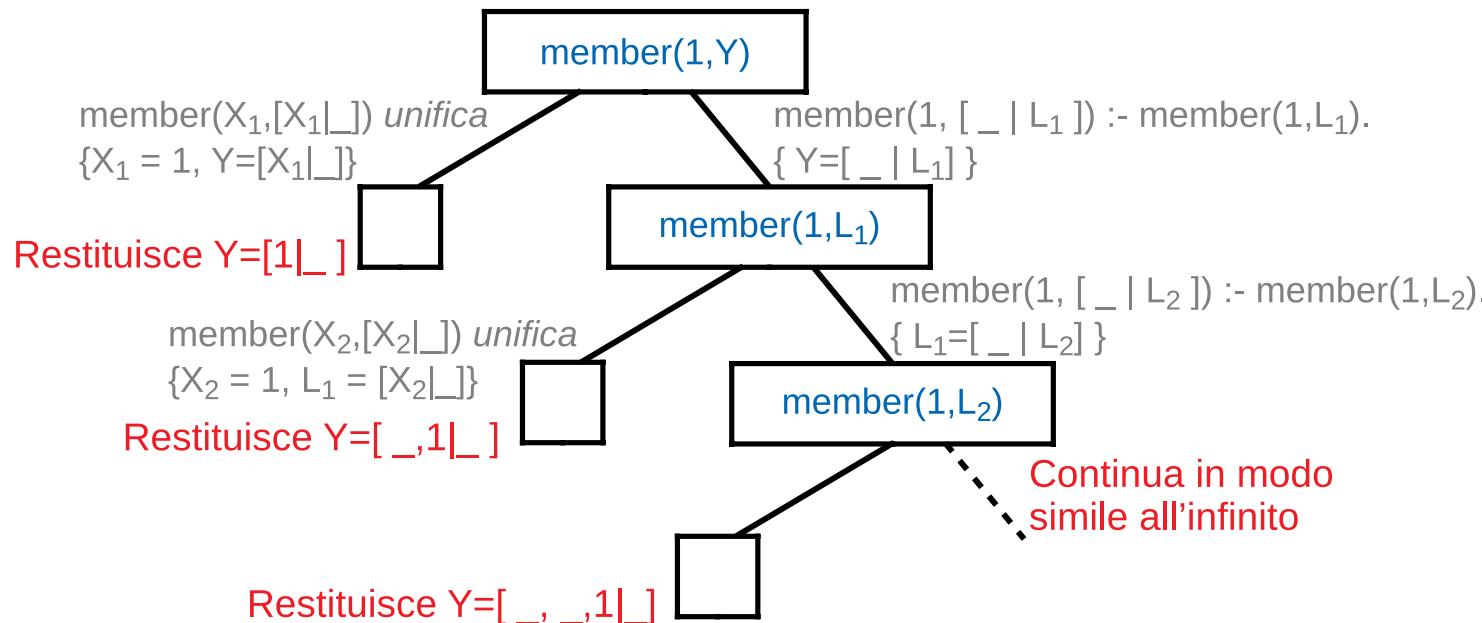
Esempi di derivazione per member



```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```



Esempi di derivazione per member



```
member(X, [X|_]).  
member(X, [_|L]) :- member(X, L).
```

Scrivete voi come si ottiene il valore di Y



Evanescenza di parametri di Input e Output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Non c'è una chiara distinzione tra parametri IN e OUT
 - ◆ Ogni parametro può essere legato a un termine con costruttori (input)



Evanescenza di parametri di Input e Output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

■ Non c'è una chiara distinzione tra parametri IN e OUT

- ◆ Ogni parametro può essere legato a un termine con costruttori (input)
- ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)



Evanescenza di parametri di Input e Output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Non c'è una chiara distinzione tra parametri IN e OUT
 - ◆ Ogni parametro può essere legato a un termine con costruttori (input)
 - ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)
 - ◆ I parametri con almeno un costruttore e una variabile forniscono sia un input sia un output



Evanescenza di parametri di Input e Output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

- Non c'è una chiara distinzione tra parametri IN e OUT
 - ◆ Ogni parametro può essere legato a un termine con costruttori (input)
 - ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)
 - ◆ I parametri con almeno un costruttore e una variabile forniscono sia un input sia un output
- Guardiamo gli esempi con member:
 - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)



Evanescenza di parametri di Input e Output

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- Non c'è una chiara distinzione tra parametri IN e OUT
 - ◆ Ogni parametro può essere legato a un termine con costruttori (input)
 - ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)
 - ◆ I parametri con almeno un costruttore e una variabile forniscono sia un input sia un output
- Guardiamo gli esempi con member:
 - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
 - ◆ `member(<elem. ground>,L)` prende un elemento e restituisce le liste che lo conengono. Modalità: (IN,OUT)



Evanescenza di parametri di Input e Output

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

- Non c'è una chiara distinzione tra parametri IN e OUT
 - ◆ Ogni parametro può essere legato a un termine con costruttori (input)
 - ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)
 - ◆ I parametri con almeno un costruttore e una variabile forniscono sia un input sia un output
- Guardiamo gli esempi con member:
 - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
 - ◆ `member(<elem. ground>,L)` prende un elemento e restituisce le liste che lo conengono. Modalità: (IN,OUT)
 - ◆ *Invertibilità dei predicati*: possono rappresentare sia una funzione sia la sua inversa



Evanescenza di parametri di Input e Output

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

- Non c'è una chiara distinzione tra parametri IN e OUT
 - ◆ Ogni parametro può essere legato a un termine con costruttori (input)
 - ◆ Ogni parametro attuale con una variabile libera produce delle sostituzioni (output)
 - ◆ I parametri con almeno un costruttore e una variabile forniscono sia un input sia un output
- Guardiamo gli esempi con member:
 - ◆ `member(X,<lista ground>)` prende una lista e restituisce i suoi elementi. Modalità: (OUT,IN)
 - ◆ `member(<elem. ground>,L)` prende un elemento e restituisce le liste che lo conengono. Modalità: (IN,OUT)
 - ◆ *Invertibilità dei predicati*: possono rappresentare sia una funzione sia la sua inversa
- Incontreremo lo stesso fenomeno in `append/3` (predicato append con 3 argomenti)



Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]

(a) scrivere una opportuna query a member che restituisce gli studenti maschi



Esercizi

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
 - (a) scrivere una opportuna query a member che restituisce gli studenti maschi
 - (b) disegnare il search tree per la query e la lista data.
 - (c) scrivere le sostituzioni di risposta

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
 - (a) scrivere una opportuna query a member che restituisce gli studenti maschi
 - (b) disegnare il search tree per la query e la lista data.
 - (c) scrivere le sostituzioni di risposta
2. Usando member, scrivere la regola per una select (algebra relazionale) sulla relazione r che seleziona i record il cui secondo elemento appartiene a una lista data. Se r ha n colonne, questa select ha n+1 argomenti:

```
select_r_2(X1, X2, ..., Xn, Lista)
```



Esercizi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione nondeterministica

Unicità di Prolog

1. Data la lista [stud(mario,m), stud(maria,f), stud(paolo,m)]
 - (a) scrivere una opportuna query a member che restituisce gli studenti maschi
 - (b) disegnare il search tree per la query e la lista data.
 - (c) scrivere le sostituzioni di risposta
2. Usando member, scrivere la regola per una select (algebra relazionale) sulla relazione r che seleziona i record il cui secondo elemento appartiene a una lista data. Se r ha n colonne, questa select ha n+1 argomenti:

```
select_r_2(X1, X2, ..., Xn, Lista)
```

3. Scrivere un predicato reverse che inverte una lista. Prendere esempio dalla soluzione per ML che fa uso di un accumulatore per costruire progressivamente la lista invertita:

```
/* significato dei parametri del predicato */  
reverse(Lista, ListaInvertita, Accumulatore)
```

Poi usare l'overloading per avere una reverse a 2 soli argomenti.



Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */  
append( Lista1, Lista2, Risultato )
```

Ispirarsi alla soluzione ricorsiva per ML

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */  
append( Lista1 , Lista2 , Risultato )
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */  
append( Lista1 , Lista2 , Risultato )
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.
 - verificare disegnando il search tree per la lista [a,b,c,d]

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */  
append( Lista1 , Lista2 , Risultato )
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.
 - verificare disegnando il search tree per la lista [a,b,c,d]
3. Usare l'invertibilità di append per verificare se [a,b,c] è un suffisso di una lista data.

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */  
append( Lista1 , Lista2 , Risultato )
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.
 - verificare disegnando il search tree per la lista [a,b,c,d]
3. Usare l'invertibilità di append per verificare se [a,b,c] è un suffisso di una lista data.
 - verificare disegnando il search tree per la lista [a,a,b,c]

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member
IN e OUT

Append

Negazione

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog



Il predicato append

1. Scrivere un predicato append che concatena due liste:

```
/* schema */  
append( Lista1 , Lista2 , Risultato )
```

Ispirarsi alla soluzione ricorsiva per ML

2. Usare l'invertibilità di append per verificare se [a,b,c] è un prefisso di una lista data.
 - verificare disegnando il search tree per la lista [a,b,c,d]
3. Usare l'invertibilità di append per verificare se [a,b,c] è un suffisso di una lista data.
 - verificare disegnando il search tree per la lista [a,a,b,c]
4. Usare l'invertibilità di append per definire un predicato last(L,X) che è vero se X è l'ultimo elemento della lista L.

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione nondeterministica

Unicità di Prolog



Ancora append

■ Generazione di tutti i prefissi / suffissi di una lista

```
prefix(Prefix, List) :- append(Prefix, _, List).
```

```
suffix(Suffix, List) :- append(_, Suffix, List).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)



Ancora append

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

Sintassi

Il predicato member

IN e OUT

Append

Negazione

[Applicazioni](#)

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

■ Generazione di tutti i prefissi / suffissi di una lista

```
prefix(Prefix, List) :- append(Prefix, _, List).
```

```
suffix(Suffix, List) :- append(_, Suffix, List).
```

■ Definizione alternativa di member attraverso append

```
member(X, L) :- append(_, [X|_], L).
```



Ancora append

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

■ Generazione di tutti i prefissi / suffissi di una lista

```
prefix(Prefix, List) :- append(Prefix, _, List).  
  
suffix(Suffix, List) :- append(_, Suffix, List).
```

■ Definizione alternativa di member attraverso append

```
member(X, L) :- append(_, [X|_], L).
```

■ Calcolo delle sottoliste di L. Notare che S è una sottolista di L sse valgono queste due condizioni (equivalenti)

- ◆ S è il prefisso di un suffisso di L
- ◆ S è il suffisso di un prefisso di L

```
sublist(Sub, L) :- prefix(Pre, List), suffix(Sub, Pre).
```



Ancora cammini su grafi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- Ora possiamo definire i cammini su un grafo, supportando anche cicli
- Definiamo il predicato `connessi_evitando(X,Y,Avoid)`: c'è un cammino da X a Y che evita i nodi nella lista Avoid

```
connessi_evitando(X,X,Avoid) :- \+ member(X,Avoid).  
connessi_evitando(X,Y,Avoid) :- arco(X,Z),  
                                \+ member(Z,Avoid),  
                                connessi_evitando(Z,Y,[X|Avoid])
```



Ancora cammini su grafi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Sintassi

Il predicato member

IN e OUT

Append

Negazione

Applicazioni

Programmazione
nondeterministica

Unicità di Prolog

- Ora possiamo definire i cammini su un grafo, supportando anche cicli
- Definiamo il predicato `connessi_evitando(X,Y,Avoid)`: c'è un cammino da X a Y che evita i nodi nella lista Avoid

```
connessi_evitando(X,X,Avoid) :- \+ member(X,Avoid).  
connessi_evitando(X,Y,Avoid) :- arco(X,Z),  
                                \+ member(Z,Avoid),  
                                connessi_evitando(Z,Y,[X|Avoid])
```

- Ora (ri)definiamo la connessione tra due nodi:

```
connessi3(X,Y) :- connessi_evitando(X,Y,[]).
```

`connessi3` termina sempre, anche su grafi dotati di cicli



[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[**Applicazioni**](#)

Derivate

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Calcolo simbolico in Prolog



Calcolo simbolico delle derivate

- In Prolog gli operatori aritmetici sono *costruttori*
 - ◆ $3+5*2$ *non* è uguale a 13!

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

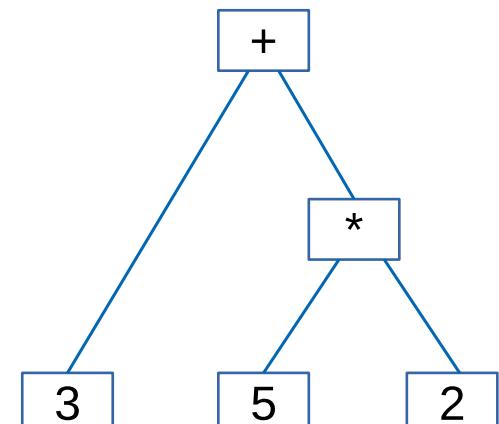
[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

In Prolog gli operatori aritmetici sono *costruttori*

- ◆ $3+5*2$ *non* è uguale a 13!
- ◆ denota invece l'albero sintattico qui a destra
- ◆ Per calcolare $3+5*2$ usare **is**:
`Ris is 3+5*2`
- ◆ questo goal è vero se Ris è uguale al *valore* di $3+5*2$





Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

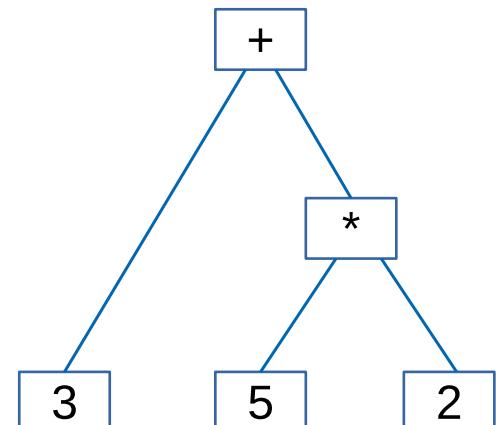
[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

In Prolog gli operatori aritmetici sono *costruttori*

- ◆ $3+5*2$ *non* è uguale a 13!
- ◆ denota invece l'albero sintattico qui a destra
- ◆ Per calcolare $3+5*2$ usare **is**:
`Ris is 3+5*2`
- ◆ questo goal è vero se Ris è uguale al *valore* di $3+5*2$



Questo rende il calcolo simbolico particolarmente semplice. Facciamo un esempio basato sulle derivate, stile Matlab



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).  
der(cos(X), X, -sen(X)).
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).  
der(cos(X), X, -sen(X)).  
der(log(X), X, 1/X).
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).  
der(cos(X), X, -sen(X)).  
der(log(X), X, 1/X).  
der(F+G, X, DF+DG) :-  
    der(F, X, DF), der(G, X, DG).
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).  
der(cos(X), X, -sen(X)).  
der(log(X), X, 1/X).  
der(F+G, X, DF+DG) :-  
    der(F, X, DF), der(G, X, DG).  
der(F-G, X, DF-DG) :-  
    der(F, X, DF), der(G, X, DG).
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).  
der(cos(X), X, -sen(X)).  
der(log(X), X, 1/X).  
der(F+G, X, DF+DG) :-  
    der(F, X, DF), der(G, X, DG).  
der(F-G, X, DF-DG) :-  
    der(F, X, DF), der(G, X, DG).  
der(F*G, X, F*Dg+Df*G) :-  
    der(F, X, DF), der(G, X, DG).  
...
```



Calcolo simbolico delle derivate

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione nondeterministica](#)

[Unicità di Prolog](#)

Schema del predicato:

der(Expr,X,D) è vero se D è la derivata di Expr rispetto a X

```
der(X, X, 1).  
der(X^N, X, N*X^N1) :-  
    N1 is N-1.  
der(sen(X), X, cos(X)).  
der(cos(X), X, -sen(X)).  
der(log(X), X, 1/X).  
der(F+G, X, DF+DG) :-  
    der(F, X, DF), der(G, X, DG).  
der(F-G, X, DF-DG) :-  
    der(F, X, DF), der(G, X, DG).  
der(F*G, X, F*Dg+Df*G) :-  
    der(F, X, DF), der(G, X, DG).  
...
```

```
?- der(x^3*cos(x), x, D).
```

```
D = x^3* -sen(x)+3*x^2*cos(x).
```

[Si possono aggiungere predicati per semplificare il risultato]



Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Usando `is` definire

1. un predicato `len(L)` che conta gli elementi della lista `L`



Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

Derivate

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Usando `is` definire

1. un predicato `len(L)` che conta gli elementi della lista `L`
2. un predicato `fatt(N,F)` tale che `F` è il fattoriale di `N`



Esercizi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Derivate](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

Usando `is` definire

1. un predicato `len(L)` che conta gli elementi della lista `L`
2. un predicato `fatt(N,F)` tale che `F` è il fattoriale di `N`
3. un predicato `sum(L,S)` tale che `S` è la somma degli elementi nella lista `L`



Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[**In che consiste**](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)



Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```



Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X



Programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X
- ◆ per ciascuno di essi verifica se è effettivamente una soluzione (test)



Programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X
- ◆ per ciascuno di essi verifica se è effettivamente una soluzione (test)
- ◆ se sì, restituisce la soluzione; altrimenti fa backtrack e torna a generare il successivo candidato



Programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ In cosa consiste:

- ◆ invece di dire a Prolog *come* trovare la soluzione di un problema
- ◆ si dice *cosa* è una soluzione e si lascia che Prolog la cerchi con il backtrack (visita del search tree)

■ Schema generale (*generate and test*):

```
solution(X) :- generate(X), test(X).
```

- ◆ Prolog genera via via i candidati a soluzione X
- ◆ per ciascuno di essi verifica se è effettivamente una soluzione (test)
- ◆ se sì, restituisce la soluzione; altrimenti fa backtrack e torna a generare il successivo candidato
- ◆ se si chiedono altre risposte, genera altre soluzioni



Confronto con gli altri paradigmi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo candidato;
while not test(X) and X ≠ null do
    X := prossimo_candidato(X)
return X
```



Confronto con gli altri paradigmi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
    X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione( X ) =
    if test(X) then X
    else soluzione( prossimo_candidato(X) )
```



Confronto con gli altri paradigmi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
    X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione( X ) =
    if test(X) then X
    else soluzione( prossimo_candidato(X) )

/* invocare così */
soluzione( primo_candidato );
```



Confronto con gli altri paradigmi

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

- La programmazione nondeterministica è una caratteristica unica del paradigma logico
- Come verrebbe approssimata negli altri paradigmi:

```
/* paradigma imperativo */
X := primo_candidato;
while not test(X) and X ≠ null do
    X := prossimo_candidato(X)
return X

/* paradigma funzionale */
fun soluzione( X ) =
    if test(X) then X
    else soluzione( prossimo_candidato(X) )

/* invocare così */
soluzione( primo_candidato );
```

- In entrambi i casi verrebbe generata solo la prima soluzione trovata (se si vogliono le altre occorre complicare il codice)
 - ◆ l'utilità del generare tutte le soluzioni sarà illustrata programmando l'AI di un gioco



Esempi di programmazione nondeterministica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

■ Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.
```



Esempi di programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.

/* invece di ricorsione ad hoc */
membro_pari(X,[X|_]) :- 0 is X mod 2.
membro_pari(X,[_|Resto]) :- membro_pari(X,Resto).
```



Esempi di programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.

/* invece di ricorsione ad hoc */
membro_pari(X,[X|_]) :- 0 is X mod 2.
membro_pari(X,[_|Resto]) :- membro_pari(X,Resto).
```

- Notare come l'approccio generate & test possa giocare il ruolo delle funzioni di ordine superiore in ML
 - ◆ permette di comporre facilmente nuovi predicati da quelli dati



Esempi di programmazione nondeterministica

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ Ricerca dei membri pari di una lista

```
/* stile generate and test */
membro_pari(X,L) :- member(X,L), 0 is X mod 2.

/* invece di ricorsione ad hoc */
membro_pari(X,[X|_]) :- 0 is X mod 2.
membro_pari(X,[_|Resto]) :- membro_pari(X,Resto).
```

■ Notare come l'approccio generate & test possa giocare il ruolo delle funzioni di ordine superiore in ML

- ◆ permette di comporre facilmente nuovi predicati da quelli dati
- ◆ in questo caso `member`, che diventa uno strumento generale per visitare una lista
- ◆ funge da *filter*: la query congiuntiva

```
member(X,Lista), predicato(X).
```

è l'analogo di: `filter predicato Lista`



Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

■ Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-  
    possible_move(Player ,Move), optimal(Player ,Move).
```



Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-  
    possible_move(Player ,Move) , optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test



Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-  
    possible_move(Player ,Move), optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
- Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco



Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-  
    possible_move(Player ,Move) , optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
- Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco
 - ◆ ad es. il primo giocatore ha una strategia vincente?



Applicazione ai giochi

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

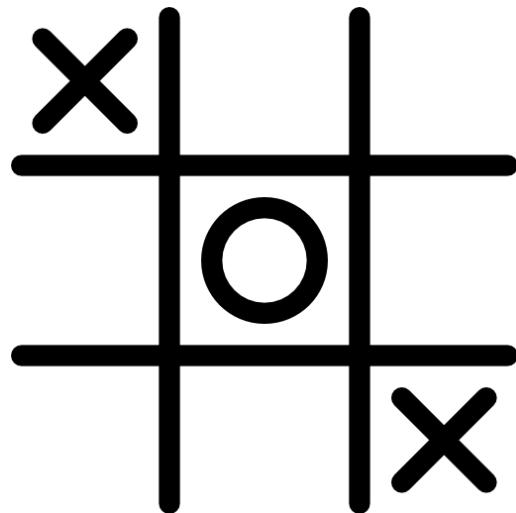
- Un possibile pattern generate-and-test per i giochi

```
next_move(Player ,Move) :-  
    possible_move(Player ,Move) , optimal(Player ,Move).
```

- A sua volta il controllo di ottimalità (cioè verificare se con Move sicuramente può vincere o almeno pareggiare) può essere effettuato con generate-and-test
- Nota: *optimal* mi dice se Move è la prima mossa di una strategia di gioco ottima → posso usare *optimal* per analizzare il gioco
 - ◆ ad es. il primo giocatore ha una strategia vincente?
 - ◆ mostrerà che è utile generare *tutte* le soluzioni



Caso di studio: Tic tac toe (aka tris)



Un semplice programma di AI che gioca a tris (imbattibile)

- una pagina di codice per “l’intelligenza”
- una pagina per i turni e l’interfaccia utente

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

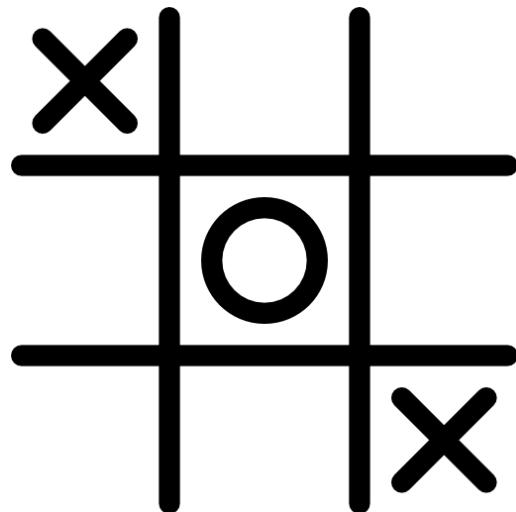
I/O elementare

GUI

[Unicità di Prolog](#)



Caso di studio: Tic tac toe (aka tris)



Un semplice programma di AI che gioca a tris (imbattibile)

- una pagina di codice per “l’intelligenza”
- una pagina per i turni e l’interfaccia utente

- Effettua una ricerca della strategia ottima:
 - ◆ ne adotta una vincente, se esiste
 - ◆ altrimenti mira al pareggio

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

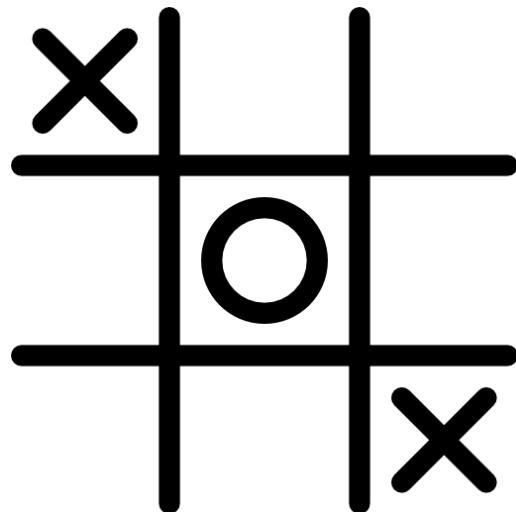
I/O elementare

GUI

[Unicità di Prolog](#)



Caso di studio: Tic tac toe (aka tris)



Un semplice programma di AI che gioca a tris (imbattibile)

- una pagina di codice per “l’intelligenza”
- una pagina per i turni e l’interfaccia utente

- Effettua una ricerca della strategia ottima:
 - ◆ ne adotta una vincente, se esiste
 - ◆ altrimenti mira al pareggio
- Approccio *generate-and-test* (programmazione nondeterministica) alla scelta della mossa. Ad ogni turno:
 1. genera una mossa
 2. verifica se è ottimale (la prima di una strategia ottima)



Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Rappresentiamo la scacchiera come una lista di 3 righe (liste):

```
start( [[1,2,3] , [4,5,6] , [7,8,9]] ).
```



Tic tac toe – Descrizione del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

Rappresentiamo la scacchiera come una lista di 3 righe (liste):

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

Introduciamo due simboli “x” e “o” per i giocatori. Quando un giocatore fa una mossa, il suo simbolo occuperà quella casella sulla scacchiera.

```
adversary(x,o).  
adversary(o,x).
```



Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Rappresentiamo la scacchiera come una lista di 3 righe (liste):

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

Introduciamo due simboli “x” e “o” per i giocatori. Quando un giocatore fa una mossa, il suo simbolo occuperà quella casella sulla scacchiera.

```
adversary(x,o).  
adversary(o,x).
```

Definiamo la condizione di vittoria:

```
win(P, [[P,P,P],_,_]).  
win(P, [_,[P,P,P],_]).  
win(P, [_,_,[P,P,P]]).
```



Tic tac toe – Descrizione del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

Rappresentiamo la scacchiera come una lista di 3 righe (liste):

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

Introduciamo due simboli “x” e “o” per i giocatori. Quando un giocatore fa una mossa, il suo simbolo occuperà quella casella sulla scacchiera.

```
adversary(x,o).  
adversary(o,x).
```

Definiamo la condizione di vittoria:

```
win(P, [[P,P,P],_,_]).  
win(P, [_,[P,P,P],_]).  
win(P, [_,_,[P,P,P]]).  
win(P, [[P,_,_],[P,_,_],[P,_,_]]).  
win(P, [[_,P,_],[_,P,_],[_,P,_]]).  
win(P, [[_,_,P],[_,_,P],[_,_,P]])
```



Tic tac toe – Descrizione del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

Rappresentiamo la scacchiera come una lista di 3 righe (liste):

```
start([[1,2,3],[4,5,6],[7,8,9]]).
```

Introduciamo due simboli “x” e “o” per i giocatori. Quando un giocatore fa una mossa, il suo simbolo occuperà quella casella sulla scacchiera.

```
adversary(x,o).  
adversary(o,x).
```

Definiamo la condizione di vittoria:

```
win(P, [[P,P,P],_,_]).  
win(P, [_,[P,P,P],_]).  
win(P, [_,_,[P,P,P]]).  
win(P, [[P,_,_],[P,_,_],[P,_,_]]).  
win(P, [[_,P,_],[_,P,_],[_,P,_]]).  
win(P, [[_,_,P],[_,_,P],[_,_,P]]).  
win(P, [[P,_,_],[_,P,_],[_,_,P]]).  
win(P, [[_,_,P],[_,P,_],[P,_,_]]).
```



Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Definiamo una situazione “non finale” (nessuno ha ancora vinto e la scacchiera non è piena):

```
non_final(Board) :-  
    \+ win(_,Board),  
    member(Row,Board),  
    member(Cell,Row),  
    number(Cell).      /* predicato built-in */
```



Tic tac toe – Descrizione del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

[Analisi del gioco](#)

[I/O elementare](#)

[GUI](#)

[Unicità di Prolog](#)

Definiamo una situazione “non finale” (nessuno ha ancora vinto e la scacchiera non è piena):

```
non_final(Board) :-  
    \+ win(_,Board),  
    member(Row,Board),  
    member(Cell,Row),  
    number(Cell).      /* predicato built-in */  
  
final(Board) :-  
    \+ non_final(Board).
```



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)`



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- **P** (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- **P** (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- **N** la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, `niente = IN-OUT` (solo un suggerimento al programmatore, data l'invertibilità)



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, `niente = IN-OUT` (solo un suggerimento al programmatore, data l'invertibilità)

```
move(P, N, Board1, Board2) :-  
    \+ win(_, Board1),
```



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

[Analisi del gioco](#)

[I/O elementare](#)

[GUI](#)

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, `niente = IN-OUT` (solo un suggerimento al programmatore, data l'invertibilità)

```
move(P, N, Board1, Board2) :-  
  \+ win(_, Board1),  
  append(RowsBefore, [Row | RowsAfter], Board1),  
  append(CellsBefore, [N | CellsAfter], Row),
```



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, `niente = IN-OUT` (solo un suggerimento al programmatore, data l'invertibilità)

```
move(P, N, Board1, Board2) :-  
  \+ win(_, Board1),  
  append(RowsBefore, [Row | RowsAfter], Board1),  
  append(CellsBefore, [N | CellsAfter], Row),  
  number(N),
```



Tic tac toe – Possibili mosse e loro effetto

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `move(+P,+N,+Board1,-Board2)` dove

- `P` (player) è il simbolo del giocatore che fa la mossa ('x' oppure 'o')
- `N` la mossa, indicata dal numero della cella ($1 \leq N \leq 9$)
- `Board1` è l'attuale scacchiera
- `Board2` è la scacchiera dopo la mossa
- Convenzione nella documentazione Prolog: `+ = IN`, `- = OUT`, `niente = IN-OUT` (solo un suggerimento al programmatore, data l'invertibilità)

```
move(P, N, Board1, Board2) :-  
  \+ win(_, Board1),  
  append(RowsBefore, [Row | RowsAfter], Board1),  
  append(CellsBefore, [N | CellsAfter], Row),  
  number(N),  
  append(CellsBefore, [P | CellsAfter], NewRow),  
  append(RowsBefore, [NewRow | RowsAfter], Board2).
```



Tic tac toe – Le strategie (generate and test)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

Schema: `has_XXX_strat(+P,-Move,+Board)` dove

- `P` (player) è il simbolo ‘x’ oppure ‘o’
- `Move` è la prima mossa della strategia
- `Board` è l’attuale scacchiera

Significato: *nella situazione descritta da Board, P ha una strategia di tipo XXX (vincente o non-perdente) che inizia con Move*



Tic tac toe – Le strategie (generate and test)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[In che consiste](#)

Tic tac toe

[Analisi del gioco](#)

[I/O elementare](#)

[GUI](#)

[Unicità di Prolog](#)

Schema: `has_XXX_strat(+P,-Move,+Board)` dove

- `P` (player) è il simbolo ‘x’ oppure ‘o’
- `Move` è la prima mossa della strategia
- `Board` è l’attuale scacchiera

Significato: *nella situazione descritta da Board, P ha una strategia di tipo XXX (vincente o non-perdente) che inizia con Move*

```
has_win_strat(P,_,Board) :-  
    win(P,Board).  
has_win_strat(P,Move,Board) :-  
    move(P,Move,Board,Board2),  
    adversary(P,Adv),  
    \+ has_tie_strat(Adv,_,Board2).
```



Tic tac toe – Le strategie (generate and test)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

Schema: `has_XXX_strat(+P,-Move,+Board)` dove

- `P` (player) è il simbolo ‘x’ oppure ‘o’
- `Move` è la prima mossa della strategia
- `Board` è l’attuale scacchiera

Significato: *nella situazione descritta da Board, P ha una strategia di tipo XXX (vincente o non-perdente) che inizia con Move*

```
has_win_strat(P,_,Board) :-  
    win(P,Board).  
has_win_strat(P,Move,Board) :-  
    move(P,Move,Board,Board2),  
    adversary(P,Adv),  
    \+ has_tie_strat(Adv,_,Board2).  
  
has_tie_strat( _,_,Board) :-  
    final(Board),  
    \+ win(_,Board).  
has_tie_strat(P,Move,Board) :-  
    move(P,Move,Board,Board2),  
    adversary(P,Adv),  
    \+ has_win_strat(Adv,_,Board2).
```



Tic-tac-toe – Analisi del gioco

- Esiste una strategia vincente per il primo giocatore?

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

[**Analisi del gioco**](#)

I/O elementare

GUI

[Unicità di Prolog](#)



Tic-tac-toe – Analisi del gioco

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

[Analisi del gioco](#)

I/O elementare

GUI

[Unicità di Prolog](#)

■ Esiste una strategia vincente per il primo giocatore?

```
?- start(Board), has_win_strat(x, Move, Board).  
false.
```

NO: nessuna mossa iniziale garantisce al primo giocatore di vincere



Tic-tac-toe – Analisi del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

- Esiste una strategia vincente per il primo giocatore?

```
?- start(Board), has_win_strat(x, Move, Board).  
false.
```

NO: nessuna mossa iniziale garantisce al primo giocatore di vincere

- Esiste una strategia vincente per il secondo giocatore?



Tic-tac-toe – Analisi del gioco

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ Esiste una strategia vincente per il primo giocatore?

```
?- start(Board), has_win_strat(x, Move, Board).  
false.
```

NO: nessuna mossa iniziale garantisce al primo giocatore di vincere

■ Esiste una strategia vincente per il secondo giocatore?

```
?- start(B0), has_tie_strat(x, Mv, B0).  
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],  
Mv = 1 ;  
...  
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],  
Mv = 9 ;  
false.
```

NO: ogni mossa iniziale permette al primo giocatore di pareggiare (se non commette errori)

- ◆ tutte le mosse iniziali fanno parte di una strategia di pareggio



Tic-tac-toe – Analisi del gioco (II)

■ Come può pareggiare il secondo giocatore?

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

[**Analisi del gioco**](#)

I/O elementare

GUI

[Unicità di Prolog](#)



Tic-tac-toe – Analisi del gioco (II)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,1,B0,B1), has_tie_strat(o,Mv,B1).  
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],  
B1 = [[x, 2, 3], [4, 5, 6], [7, 8, 9]],  
Mv = 5 ;  
false.
```

Se la prima mossa è su un angolo, l'unica risposta è 5, in tutti gli altri casi vince il primo giocatore



Tic-tac-toe – Analisi del gioco (II)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

■ Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,1,B0,B1), has_tie_strat(o,Mv,B1).  
B0 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]],  
B1 = [[x, 2, 3], [4, 5, 6], [7, 8, 9]],  
Mv = 5 ;  
false.
```

Se la prima mossa è su un angolo, l'unica risposta è 5, in tutti gli altri casi vince il primo giocatore

```
?- start(B0), move(x,2,B0,B1), has_tie_strat(o,Mv,B1).  
...  
Mv = 1 ;  
...  
Mv = 3 ;  
...  
Mv = 5 ;  
...  
Mv = 8 ;  
false.
```

Quindi se la prima mossa è 2, meglio non rispondere con 4, 6, 7 o 9!



Tic-tac-toe – Analisi del gioco (III)

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

[Analisi del gioco](#)

I/O elementare

GUI

[Unicità di Prolog](#)

■ Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,5,B0,B1), has_tie_strat(o,Mv,B1).
```



Tic-tac-toe – Analisi del gioco (III)

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

- Come può pareggiare il secondo giocatore?

```
?- start(B0), move(x,5,B0,B1), has_tie_strat(o,Mv,B1).  
...  
Mv = 1 ;  
...  
Mv = 3 ;  
...  
Mv = 7 ;  
...  
Mv = 9 ;  
false.
```

Se la prima mossa è 5, si deve scegliere una casella d'angolo

- Tutte queste affermazioni si possono verificare empiricamente forzando il primo passo:

```
?- start(B0), move(x,5,B0,B1), turn(user,o,B1).
```



Tic tac toe – L'interfaccia utente testuale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

(per le interfacce grafiche vedere l'estensione XPCE,
<https://www.swi-prolog.org/packages/xpce/>)

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```



Tic tac toe – L'interfaccia utente testuale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

(per le interfacce grafiche vedere l'estensione XPCE,
<https://www.swi-prolog.org/packages/xpce/>)

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```

```
print_board([]).
print_board([Row|Rest]) :-
    format('~a|~a|~a\n', Row),
    print_board(Rest).
```



Tic tac toe – L'interfaccia utente testuale

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

(per le interfacce grafiche vedere l'estensione XPCE,
<https://www.swi-prolog.org/packages/xpce/>)

```
cpu move: 1
x|2|3
4|5|6
7|8|9

Player o insert your move [1-9]: 8
x|2|3
4|5|6
7|o|9
```

```
print_board([]).
print_board([Row|Rest]) :-
    format('`~a|`~a|`~a`\n', Row),
    print_board(Rest).

read_move(Player,Move) :-
    format('\nPlayer `~a insert your move [1-9]: ', [Player]),
    get_single_char(Char), put_char(Char), nl,
    Move is Char-48,
    Move >= 1, Move =< 9.
```



Tic tac toe – I turni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
turn(_,_,Board) :-  
    final(Board),  
    \+ win(_,Board),  
    format('`The game ends in a tie.\n').
```



Tic tac toe – I turni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
turn(_,_,Board) :-  
    final(Board),  
    \+ win(_,Board),  
    format('`The game ends in a tie.\n').
```

```
turn(P,_,Board) :-  
    win(_,Board),  
    member(Adv,[user,cpu]), Adv \= P,  
    format(`The ~a wins!\n', [Adv]).
```



Tic tac toe – I turni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
turn(_,_,Board) :-  
    final(Board),  
    \+ win(_,Board),  
    format('`The game ends in a tie.\n').  
  
turn(P,_,Board) :-  
    win(_,Board),  
    member(Adv,[user,cpu]), Adv \= P,  
    format(`The ~a wins!\n', [Adv]).  
  
turn(user,P,Board) :-  
    read_move(P,M),  
    move(P, M, Board, Board2),  
    print_board(Board2),  
    adversary(P,Adv),  
    turn(cpu,Adv,Board2).
```



Tic tac toe – I turni

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
turn(_,_,Board) :-  
    final(Board),  
    \+ win(_,Board),  
    format('`The game ends in a tie.\n').  
  
turn(P,_,Board) :-  
    win(_,Board),  
    member(Adv,[user,cpu]), Adv \= P,  
    format(`The ~a wins!\n', [Adv]).  
  
turn(user,P,Board) :-  
    read_move(P,M),  
    move(P, M, Board, Board2),  
    print_board(Board2),  
    adversary(P,Adv),  
    turn(cpu,Adv,Board2).  
  
turn(cpu,P,Board) :-  
    (has_win_strat(P,Move,Board); has_tie_strat(P,Move,Board)),  
    format(`cpu move: ~a\n', [Move]),  
    move(P, Move, Board, Board2),  
    print_board(Board2),  
    adversary(P,Adv),  
    turn(user,Adv,Board2).
```



Tic-tac-toe – II “main”

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)



Tic-tac-toe – II “main”

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

```
go(First ,Symbol) :-  
    member(First ,[user ,cpu]) ,  
    start(Board) ,  
    turn(First ,Symbol ,Board).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)



Tic-tac-toe – II “main”

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

```
go(First ,Symbol) :-  
    member(First ,[user ,cpu]) ,  
    start(Board) ,  
    turn(First ,Symbol ,Board).
```

Esempio di invocazione

```
?- go(user ,x).
```

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)



Tic-tac-toe – II “main”

Paradigma logico

Prolog

Costrutti Prolog

Unificazione

Conjunctive queries

Le Regole

Ragionamento

Liste

Applicazioni

Programmazione
nondeterministica

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

Unicità di Prolog

Due modalità di esecuzione:

- Specificando liberamente chi inizia e che simbolo usa

```
go(First,Symbol) :-  
    member(First,[user,cpu]),  
    start(Board),  
    turn(First,Symbol,Board).
```

Esempio di invocazione

```
?- go(user,x).
```

- Estrazione a sorte di chi inizia

```
main :-  
    choice is random(2),  
    nth0(choice,[user,o],(cpu,x)),(First,Symbol)),  
    format('The ~a starts\n', [First]),  
    start(Board),  
    turn(First,Symbol,Board),  
    halt.
```

Demo



Tic-tac-toe – Compilazione in codice stand-alone

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
swipl --goal=main --stand_alone=true -o tic-tac-toe  
      -c tic-tac-toe.pl
```

■ Spiegazione delle opzioni:

- ◆ **--stand_alone**: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)



Tic-tac-toe – Compilazione in codice stand-alone

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
swipl --goal=main --stand_alone=true -o tic-tac-toe  
      -c tic-tac-toe.pl
```

■ Spiegazione delle opzioni:

- ◆ --stand_alone: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- ◆ -o: nome del file oggetto



Tic-tac-toe – Compilazione in codice stand-alone

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
swipl --goal=main --stand_alone=true -o tic-tac-toe  
      -c tic-tac-toe.pl
```

■ Spiegazione delle opzioni:

- ◆ --stand_alone: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- ◆ -o: nome del file oggetto
- ◆ -c: nome del file sorgente



Tic-tac-toe – Compilazione in codice stand-alone

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
swipl --goal=main --stand_alone=true -o tic-tac-toe  
      -c tic-tac-toe.pl
```

■ Spiegazione delle opzioni:

- ◆ --stand_alone: crea un codice direttamente eseguibile (invece di far partire la macchina virtuale)
- ◆ -o: nome del file oggetto
- ◆ -c: nome del file sorgente
- ◆ --goal: il goal da invocare quando il file viene eseguito



Cenni a una possibile interfaccia grafica con XPCE





Cenni a una possibile interfaccia grafica con XPCE



```
gui :-  
  /* costruzione buttoni */  
  
  new(Msg,button('Your turn')) ,  
  new(B1,button('')) ,  
  ...  
  new(B9,button('')) ,
```



Cenni a una possibile interfaccia grafica con XPCE



```
gui :-  
  /* costruzione buttoni */  
  
  new(Msg,button('Your_turn')),  
  new(B1,button('')),  
  ...  
  new(B9,button('')),
```

```
/* costruzione finestra */  
new(Dialog, dialog('Tic_Tac_Toe')),  
send(Dialog, gap, size(10,10)),  
send(Dialog, append, B1),  
send(Dialog, append, B2, right),  
send(Dialog, append, B3, right),  
send(Dialog, append, B4, below),  
...  
send(Dialog, append, Msg, below),  
send(Msg, alignment, left),  
send(Dialog, open),  
/* il gioco inizia */  
gui_go(Msg,Buttons).
```



Cenni a una possibile interfaccia grafica con XPCE



```
gui :-  
  /* costruzione bottoni */  
  
  new(Msg,button('Your\turn')),  
  new(B1,button('')),  
  ...  
  new(B9,button('')),
```

```
/* costruzione finestra */  
new(Dialog, dialog('Tic\Tac\Toe')),  
send(Dialog, gap, size(10,10)),  
send(Dialog, append, B1),  
send(Dialog, append, B2, right),  
send(Dialog, append, B3, right),  
send(Dialog, append, B4, below),  
...  
send(Dialog, append, Msg, below),  
send(Msg, alignment, left),  
send(Dialog, open),  
/* il gioco inizia */  
gui_go(Msg,Buttons).
```

```
/* inizializzazione bottoni */  
send(Bi, message,  
  message(@prolog,click,I,B1,B2,...)),  
send(Bi, label, ''),  
send(Bi, size, size(45,45))  
...  
  
/* quando si clicca */  
send(Bi, message,  
  message(@prolog,true),  
send(Bi, label, 'X')),  
...
```



Compilazione stand-alone della versione grafica

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

In che consiste

Tic tac toe

Analisi del gioco

I/O elementare

GUI

[Unicità di Prolog](#)

```
swipl --goal=gui --stand_alone=true --pce -o tic-tac-toe-gui  
      -c tic-tac-toe-gui.pl
```

■ Spiegazione delle opzioni:

- ◆ --pce rende disponibili le primitive per la GUI
- ◆ le altre sono come nell'esempio precedente

■ Attenzione! Se si usa '@' bisogna dichiararlo come operatore unario prefisso (altrimenti viene generato un syntax error)

```
/* inserire all'inizio del programma */  
:- op(1, fx, @).
```



Riassumendo: caratteristiche uniche di Prolog

■ Invertibilità dei predicati

- ◆ un singolo predicato implementa molte funzioni
- ◆ dovuto al fatto che le variabili logiche sono IN/OUT/IN-OUT a seconda dei parametri attuali

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)



Riassumendo: caratteristiche uniche di Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ Invertibilità dei predicati

- ◆ un singolo predicato implementa molte funzioni
- ◆ dovuto al fatto che le variabili logiche sono IN/OUT/IN-OUT a seconda dei parametri attuali

■ Programmazione nondeterministica

- ◆ con ricerca automatica delle soluzioni
- ◆ basato sul backtracking (cioè il meccanismo di visita dei search tree)



Riassumendo: caratteristiche uniche di Prolog

[Paradigma logico](#)

[Prolog](#)

[Costrutti Prolog](#)

[Unificazione](#)

[Conjunctive queries](#)

[Le Regole](#)

[Ragionamento](#)

[Liste](#)

[Applicazioni](#)

[Programmazione
nondeterministica](#)

[Unicità di Prolog](#)

■ Invertibilità dei predicati

- ◆ un singolo predicato implementa molte funzioni
- ◆ dovuto al fatto che le variabili logiche sono IN/OUT/IN-OUT a seconda dei parametri attuali

■ Programmazione nondeterministica

- ◆ con ricerca automatica delle soluzioni
- ◆ basato sul backtracking (cioè il meccanismo di visita dei search tree)

■ Restano molti aspetti interessanti che purtroppo non abbiamo il tempo di illustrare

- ◆ strutture dati parziali (permettono append e creazione dizionari in tempo *costante*)
- ◆ meta-predicati / reflection
- ◆ aggregati (setof)
- ◆ forall ...

Linguaggi di Programmazione I – Lezione 4

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

6 aprile 2022



Panoramica della lezione

Parametrizzazione di procedure

Bibliografia



Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Parametrizzazione di procedure



Parametri

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.

[Parametrizzazione di procedure](#)

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

[Bibliografia](#)



Parametri

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Alcuni linguaggi distinguono tre categorie:
 - ◆ parametri di ingresso (IN): sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;

[Parametrizzazione di procedure](#)

Parametri

[Tipi dei p.](#)

[Associazione dei p.](#)

[Esempio](#)

[Associazione di default](#)

[Parametri IN](#)

[Parametri OUT](#)

[Parametri IN-OUT](#)

[Esempi](#)

[Aliasing](#)

[Procedure come . . .](#)

[Macro](#)

[Esercizio](#)

[Funzioni](#)

[Bibliografia](#)



Parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Alcuni linguaggi distinguono tre categorie:
 - ◆ parametri di ingresso (IN): sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
 - ◆ parametri di uscita (OUT): sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;



Parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Alcuni linguaggi distinguono tre categorie:
 - ◆ parametri di ingresso (IN): sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
 - ◆ parametri di uscita (OUT): sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;
 - ◆ parametri sia di ingresso che di uscita (IN-OUT): servono a far transitare le informazioni in entrambe le direzioni.



Parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Alcuni linguaggi distinguono tre categorie:
 - ◆ parametri di ingresso (IN): sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
 - ◆ parametri di uscita (OUT): sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;
 - ◆ parametri sia di ingresso che di uscita (IN-OUT): servono a far transitare le informazioni in entrambe le direzioni.
- Devono essere specificati in due punti:



Parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Alcuni linguaggi distinguono tre categorie:
 - ◆ parametri di ingresso (IN): sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
 - ◆ parametri di uscita (OUT): sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;
 - ◆ parametri sia di ingresso che di uscita (IN-OUT): servono a far transitare le informazioni in entrambe le direzioni.
- Devono essere specificati in due punti:
 - ◆ nella definizione della procedura: *parametri formali*;



Parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- Costituiscono il mezzo attraverso il quale le informazioni transitano esplicitamente tra l'unità chiamante e quella chiamata.
- Alcuni linguaggi distinguono tre categorie:
 - ◆ parametri di ingresso (IN): sono passati dalla unità chiamante alla unità chiamata al momento dell'invocazione;
 - ◆ parametri di uscita (OUT): sono passati dall'unità chiamata alla unità chiamante al momento della terminazione della prima;
 - ◆ parametri sia di ingresso che di uscita (IN-OUT): servono a far transitare le informazioni in entrambe le direzioni.
- Devono essere specificati in due punti:
 - ◆ nella definizione della procedura: *parametri formali*;
 - ◆ nelle invocazioni della procedura: *parametri attuali o argomenti*.



Tipi dei parametri

- **Linguaggi staticamente tipati:** nella definizione deve essere specificato il tipo dei parametri formali; nell'invocazione è richiesta la compatibilità di tipo tra parametri formali e attuali (es.: Java, C, C++, etc.)

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia



Tipi dei parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- **Linguaggi staticamente tipati:** nella definizione deve essere specificato il tipo dei parametri formali; nell'invocazione è richiesta la compatibilità di tipo tra parametri formali e attuali (es.: Java, C, C++, etc.)
- **Linguaggi dinamicamente tipati:** i parametri formali non hanno alcun vincolo di tipo; il legame di tipo si instaura durante l'esecuzione (run time) allo stesso tipo dei parametri attuali (impossibile il type checking in compilazione) (es.: Python)



Tipi dei parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

- **Linguaggi staticamente tipati:** nella definizione deve essere specificato il tipo dei parametri formali; nell'invocazione è richiesta la compatibilità di tipo tra parametri formali e attuali (es.: Java, C, C++, etc.)
- **Linguaggi dinamicamente tipati:** i parametri formali non hanno alcun vincolo di tipo; il legame di tipo si instaura durante l'esecuzione (run time) allo stesso tipo dei parametri attuali (impossibile il type checking in compilazione) (es.: Python)



Associazione dei parametri

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Come si stabilisce la corrispondenza tra parametri attuali e formali

Due **metodi di associazione**:

- *per posizione*: a seconda della posizione relativa nella sequenza dei parametri;
- *per nome*: il nome del parametro formale è aggiunto come prefisso al parametro attuale;



Esempio

Data l'intestazione della seguente procedura (linguaggio ADA):

```
procedure TEST (A: in Atyp; b: in out Btype; C: out Ctype)
```

[Parametrizzazione di procedure](#)

[Parametri
Tipi dei p.](#)

[Associazione dei p.](#)

Esempio

[Associazione di default](#)

[Parametri IN](#)

[Parametri OUT](#)

[Parametri IN-OUT](#)

[Esempi](#)

[Aliasing](#)

[Procedure come . . .](#)

[Macro](#)

[Esercizio](#)

[Funzioni](#)

[Bibliografia](#)



Esempio

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Data l'intestazione della seguente procedura (linguaggio ADA):

```
procedure TEST (A: in Atyp; b: in out Btype; C: out Ctype)
```

allora una invocazione che usi associazione per posizione è:

```
TEST(X, Y, Z);
```



Esempio

Data l'intestazione della seguente procedura (linguaggio ADA):

```
procedure TEST (A: in Atyp; b: in out Btype; C: out Ctype)
```

allora una invocazione che usi associazione per posizione è:

```
TEST(X, Y, Z);
```

mentre una che usi associazione per nome può essere:

```
TEST(A=>X, C=>Z, b=>Y);
```

[Parametrizzazione di procedure](#)

[Parametri](#)

[Tipi dei p.](#)

[Associazione dei p.](#)

Esempio

[Associazione di default](#)

[Parametri IN](#)

[Parametri OUT](#)

[Parametri IN-OUT](#)

[Esempi](#)

[Aliasing](#)

[Procedure come ...](#)

[Macro](#)

[Esercizio](#)

[Funzioni](#)

[Bibliografia](#)



Associazione di default

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Un'ulteriore tecnica è la cosiddetta *associazione di default*

Essa permette di specificare valori di default per quei parametri formali che non sono stati legati a valori da parametri attuali.

Esempio in C++:

```
void print_error_msg(int line,  
                     string message = "Generic error")  
{ ... }
```

Questa funzione si può invocare con due argomenti o con un argomento solo (line)



Parametri IN

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. Per riferimento

- la locazione del parametro attuale diventa la locazione del parametro formale
- si deve impedire (a tempo di compilazione, se possibile) la modifica all'interno della procedura



Parametri IN

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. Per riferimento

- la locazione del parametro attuale diventa la locazione del parametro formale
- si deve impedire (a tempo di compilazione, se possibile) la modifica all'interno della procedura

2. Per copia

- il valore del parametro attuale viene copiato in una nuova locazione, quella del parametro formale



Parametri IN

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. Per riferimento

- la locazione del parametro attuale diventa la locazione del parametro formale
- si deve impedire (a tempo di compilazione, se possibile) la modifica all'interno della procedura

2. Per copia

- il valore del parametro attuale viene copiato in una nuova locazione, quella del parametro formale
- parametro formale visto come variabile locale



Parametri IN

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. Per riferimento

- la locazione del parametro attuale diventa la locazione del parametro formale
- si deve impedire (a tempo di compilazione, se possibile) la modifica all'interno della procedura

2. Per copia

- il valore del parametro attuale viene copiato in una nuova locazione, quella del parametro formale
- parametro formale visto come variabile locale
- modifica permessa, perché valida solo nell'ambiente di esecuzione della procedura.



Parametri IN

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati in due modi:

1. Per riferimento

- la locazione del parametro attuale diventa la locazione del parametro formale
- si deve impedire (a tempo di compilazione, se possibile) la modifica all'interno della procedura

2. Per copia

- il valore del parametro attuale viene copiato in una nuova locazione, quella del parametro formale
- parametro formale visto come variabile locale
- modifica permessa, perché valida solo nell'ambiente di esecuzione della procedura.

Il secondo modo è meno efficiente del primo, sia rispetto allo spazio sia al tempo, ma garantisce automaticamente che il parametro attuale non sia modificato



Parametri OUT

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati:

1. Per riferimento
2. Per copia (cioè per *risultato*)

■ La copia avviene all'uscita



Parametri OUT

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Possono essere realizzati:

1. Per riferimento
2. Per copia (cioè per *risultato*)
 - La copia avviene all'uscita

Rappresentano risultati ⇒ bisogna proibirne la “lettura”, ad es.

- uso a destra di un assegnamento
- passaggio a un parametro IN o IN OUT di un'altra procedura

Non esistono regole generali, nemmeno tra diverse versioni di uno stesso linguaggio

- Ada 83 proibisce di “leggere” i parametri OUT
- Le versioni successive invece lo permettono



Parametri IN-OUT

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Sono la combinazione dei due precedenti.



Parametri IN-OUT

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Sono la combinazione dei due precedenti. Anch'essi possono essere realizzati:

1. Per riferimento

- non ci sono limitazioni all'uso all'interno della procedura



Parametri IN-OUT

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Sono la combinazione dei due precedenti. Anch'essi possono essere realizzati:

1. Per riferimento

- non ci sono limitazioni all'uso all'interno della procedura

2. Per copia (*per valore-risultato*)

- avvengono *due processi di copia*: uno durante l'attivazione ed uno al termine della procedura



Esempi

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

| Linguaggio | IN | OUT | IN-OUT |
|---------------------|----|-----|----------------------------------|
| Ada | Sì | Sì | Sì |
| Pascal | Sì | No | Sì (keyword var) |
| C | Sì | No | No. Emulato con puntatori |
| C++ | Sì | No | Sì (con riferimenti) |
| Java ¹ | Sì | No | No. Emulato con oggetti mutabili |
| Python ² | Sì | No | No. Emulato con oggetti mutabili |

Nota 1: in Java, tutti i parametri, primitivi e non, vengono passati in modalità “IN per valore”, ma le variabili di tipo non primitivo sono riferimenti.

Nota 2: in Python, tutte le variabili sono riferimenti, e vengono passati in modalità “IN per valore”.



Aliasing

È la possibilità di riferirsi alla stessa locazione con nomi diversi.

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia



Aliasing

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

È la possibilità di riferirsi alla stessa locazione con nomi diversi. Nel passaggio dei parametri può causare notevoli problemi di interpretazione.



Aliasing

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

È la possibilità di riferirsi alla stessa locazione con nomi diversi. Nel passaggio dei parametri può causare notevoli problemi di interpretazione. Per esempio:

```
program MAIN;
  var
    A: integer;
  procedure TEST (var X, Y: integer);
  begin
    X := A + Y;
    writeln(A, X, Y)
  end;
begin
  A := 1;
  TEST(A, A)
end.
```

Esercizio: determinare l'uscita del programma nel caso in cui i parametri VAR siano realizzati *per riferimento* e nel caso in cui siano realizzati *per copia*.



Procedure come parametri di procedura

[Parametrizzazione di procedure](#)

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

[Bibliografia](#)

Alcuni linguaggi permettono l'uso di procedure come argomento di altre procedure.



Procedure come parametri di procedura

Parametrizzazione di
procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di
default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Alcuni linguaggi permettono l'uso di procedure come argomento di altre procedure. Esempio:

```
program MAIN;
  VAR a: real;
  procedure TESTPOS (X: real; procedure ERROR (MSG: string));
    begin
      if X <= 0 then ERROR ('Negative X in TESTPOS')
    end;
  procedure E1 (M: string);
    begin
      writeln('E1 error: ', M)
    end;
  procedure E2 (M: string);
    begin
      writeln('E2 error: ', M)
    end;
begin
  readln (A);
  TESTPOS(A, E1);
  TESTPOS(A, E2)
end.
```



Macro

Generazione di un nuovo brano di codice sorgente (espansione della macro) in cui i nomi dei parametri attuali sostituiscono i nomi dei parametri formali.

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia



Macro

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Generazione di un nuovo brano di codice sorgente (espansione della macro) in cui i nomi dei parametri attuali sostituiscono i nomi dei parametri formali.

Esempio: data la procedura

```
procedure swap (a, b: integer);
  var temp: integer;
  begin
    temp := a;
    a := b;
    b := temp
  end;
```

allora la chiamata swap(x, y) esegue il seguente brano di codice:

```
temp := x;
x := y;
y := temp;
```



Esercizio

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Determinare i problemi che nascono dai due programmi, se swap è una macro:

```
program main;
var
  i: integer;
  m: array [1..100] of integer;
  ...
begin
  ...
  swap(i, m[i]);
  ...
end.
```



Esercizio

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come ...

Macro

Esercizio

Funzioni

Bibliografia

Determinare i problemi che nascono dai due programmi, se swap è una macro:

```
program main;
var
  i: integer;
  m: array [1..100] of integer;
  ...
begin
  ...
  swap(i, m[i]);
  ...
end.
```

```
program main;
var
  i, temp: integer;
  ...
begin
  ...
  swap(i, temp);
  ...
end.
```



Funzioni

Sono procedure che restituiscono un valore alla procedura chiamante.

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia



Funzioni

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Sono procedure che restituiscono un valore alla procedura chiamante.
Sono realizzate

- o creando una pseudovariabile nell'ambiente locale della procedura chiamata. Tale variabile può essere solo modificata; non è possibile l'accesso in lettura.



Funzioni

Parametrizzazione di procedure

Parametri

Tipi dei p.

Associazione dei p.

Esempio

Associazione di default

Parametri IN

Parametri OUT

Parametri IN-OUT

Esempi

Aliasing

Procedure come . . .

Macro

Esercizio

Funzioni

Bibliografia

Sono procedure che restituiscono un valore alla procedura chiamante.
Sono realizzate

- o creando una pseudovariabile nell'ambiente locale della procedura chiamata. Tale variabile può essere solo modificata; non è possibile l'accesso in lettura.
- o utilizzando una istruzione di `return` per restituire esplicitamente il controllo alla procedura chiamante inviandole allo stesso tempo il valore di una espressione.



Bibliografia

Parametrizzazione di
procedure

Bibliografia

Bibliografia

Capitolo 9 (“Astrarre sul controllo”) di *Linguaggi di Programmazione, principi e paradigmi*, di Gabbrielli e Martini (2a edizione)

Polimorfismo parametrico vs polimorfismo per inclusione

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - ◆ Push(element)
 - ◆ Pop()
 - ◆ Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - ◆ Implementazione a vettore

Una soluzione

- Sfruttare Object

```
public class ObjectStack {  
  
    private Object[] stack;  
    private int top = -1;  
  
    public ObjectStack( int dim ) { stack = new Object [dim]; }  
  
    public void push( Object el ) { stack[++top] = el; }  
  
    public Object pop() { return stack[top--]; }  
}
```

Una soluzione

■ Esempio di uso #1

```
ObjectStack so = new ObjectStack( 10 );  
  
so.push(1);    // conversione implicita a Integer (autoboxing)  
so.push(2);    // e upcast automatico a Object  
so.push(3);  
  
for( int i = 0; i<3; i++ ){  
    System.out.println( ((Integer) so.pop()) . intValue() ); }  
}
```

Se non faccio il downcast...

non posso usare questo metodo
(specifico di Integer)

3
2
1

output

Una soluzione

■ Esempio di uso #2

```
ObjectStack so = new ObjectStack( 10 );  
  
so.push(1);    // conversione implicita a Integer (autoboxing)  
so.push(2);    // e upcast automatico a Object  
so.push("A");  
  
for( int i = 0; i<3; i++ ){  
    System.out.println( ((Integer) so.pop()) . intValue() ); }  
}
```

Se mi distraggo compila ancora ma...

output

```
java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
```

Una soluzione migliore: template

- Ovvero una **classe parametrica**

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int dim ) { stack = new ElemType[ dim ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Parametro formale di tipo

Ci piacerebbe...
ma non si può fare!

Una soluzione migliore: template

■ Esempio di uso #1

```
GenericStack< Integer > si = new GenericStack< Integer >( 10 );
```

```
si.push(1);
si.push(2);
si.push(3);
for( int i = 0; i<3; i++ ) {
    System.out.println( si.pop().intValue() );
}
```

Parametro attuale di tipo

Non serve downcast:
pop() restituisce Integer

3
2
1

output

Una soluzione migliore: template

■ Esempio di uso #2

```
GenericStack< Integer > si = new GenericStack< Integer >( 10 );
```

```
si.push(1);
si.push(2);
si.push("A");
for( int i = 0; i<3; i++ ) {
    System.out.println( si.pop().intValue() );
}
```

push(Integer)

1. ERROR in mylists/Test2.java (at line 8)

```
si.push("A");
^^^^
```

output del compilatore!

The method push(Integer) in the type GenericStack<Integer> is not applicable
for the arguments (String)

Confronto

- Pro del polimorfismo parametrico:
 - Non richiede controlli di tipo a run time
 - Anticipa scoperta errori di tipo a tempo di compilazione
- Pro del polimorfismo per inclusione
 - Permette strutture dati eterogenee
 - Ad esempio, Stack di elementi di tipo diverso

Prendere il meglio

- In molti casi
 - Servono collezioni di elementi eterogenei
 - Ma vogliamo usarli allo stesso modo
- Esempio:
 - Una scena è una *lista* di forme eterogenee (rettangoli, ellissi, linee, ecc.)
 - Vogliamo usarla per implementare *refresh*, che deve solo inviare un messaggio *draw()* a tutte le forme della lista
- Quindi
 - Identificare le modalità d'uso degli elementi
 - Scegliere una superclasse comune (o fattorizzarle in una *interfaccia*)
 - Usare la superclasse/interfaccia come argomento del template

Note sull'implementazione

- In Java – dopo la compilazione - diventa comunque uno stack di Object
 - I parametri di tipo vengono usati per il type checking e poi **cancellati (erasure)**
 - Per questo un parametro attuale di tipo non può essere primitivo come int o float, ma dobbiamo usare i wrapper
 - Per questo non si può istanziare un array di un tipo parametrico

Note sull'implementazione

- In C++, ogni uso di un template fa compilare una nuova classe
 - Sorta di macro
 - Il compilatore inserisce nel programma la definizione di classe specializzata e la ricompila per ogni parametro attuale
 - Più espressivo ma anche più “costoso”
 - Si può usare un parametro di tipo in tutti i modi in cui si potrebbe usare un tipo vero

Array parametrici in Java

- Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack( int dim ) { stack = (ElemType[]) new Object[ dim ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Siamo costretti a un downcast

- Otteniamo comunque un warning a tempo di compilazione

Array parametrici in Java

- Per riuscire a compilare:

```
public class GenericStack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    @SuppressWarnings("unchecked")  
    public GenericStack( int dim ) { stack = (ElemType[]) new Object[ dim ]; }  
  
    public void push( ElemType el ) { stack[++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Annotazione

- Ora niente warning
- Meglio ancora: usare ArrayList<ElemType>

Astrarre un'API tramite interfaccia

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - Push(element)
 - Pop()
 - Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - **Non “forzare” una specifica implementazione**
 - **Se ne occupa un altro team**
 - **Serve un modo per far lavorare i due team indipendentemente ma garantendo l'integrazione dei risultati**

Soluzione: interfacce

- Prima parte

```
public interface Stack<ElemType> {  
    public void push( ElemType el );  
    public ElemType pop();  
}
```

Implementare l'interfaccia

- Con un template specifico liberamente definito

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top = -1;  
  
    public GenericStack ( int dim ) { ... }  
  
    public void push( ElemType el ) { stack [++top] = el; }  
  
    public ElemType pop() { return stack[top--]; }  
}
```

Utilizzare l'interfaccia

■ Esempio

```
Stack< Integer > si = new GenericStack< Integer >( 10 );  
  
si.push(1);  
si.push(2);  
si.push(3);  
for( int i = 0; i<3; i++ ) {  
    System.out.println( si.pop().intValue() );  
}
```

Posso cambiare implementazione
semplicemente cambiando qs tipo.
Il resto del codice resta uguale.

3
2
1

output

Confronto con classi astratte

- Se avessi definito Stack come una classe astratta...
 - ...GenericStack avrebbe **dovuto estendere** Stack
 - In Java, una classe può estendere una sola altra classe
 - Restringe l'insieme di classi che si possono utilizzare
- Vantaggi delle interfacce
 - Non impone lo stesso obbligo
 - Le classi che implementano l'interfaccia Stack possono essere ovunque nella gerarchia delle classi
- Vantaggi delle classi astratte
 - Posso contenere dati (aka *stato*)

Stack con eccezioni

Esercizio

- Definire il tipo di dato “Stack” con operazioni
 - Push(element)
 - Pop()
 - Non “forzare” un tipo specifico per gli elementi
 - È una libreria: non sapete come la useranno
 - Non “forzare” una specifica implementazione
 - **Gestire pulitamente gli errori specifici**
 - **Pop su stack vuoto**

Soluzione: eccezioni

- Prima parte

```
public interface Stack<ElemType> {  
    public void push( ElemType el );  
    public ElemType pop();  
}
```

```
public class EmptyStackException extends RuntimeException {  
    public EmptyStackException() {}  
    public EmptyStackException(String msg) { super(msg); }  
}
```

Implementare l'interfaccia

- Con un template specifico liberamente definito

```
public class GenericStack<ElemType> implements Stack<ElemType> {  
    private ElemType[] stack;  
    private int top;  
  
    public GenericStack ( int dim ) { ... }  
  
    public void push( ElemType el ) { stack [++top] = el; }  
  
    public ElemType pop() {  
        if( top >= 0 ) return stack[ top --];  
        else throw new EmptyStackException();  
    }  
}
```

Utilizzo

■ Esempio di svuotamento stack

```
Stack< Integer > si = new GenericStack< Integer >();
```

```
si.push(1);
si.push(2);
si.push(3);
try {
    while( true ) {
        System.out.println( si.pop().intValue() );
    }
} catch( RuntimeException e ) {
    System.out.println( "fine" );
}
```

Non serve downcast:
pop() restituisce Integer

3
2
1
fine

output

Linguaggi di Programmazione I – Lezione 3

Prof. Marco Faella

<mailto://m.faella@unina.it>

<http://wpage.unina.it/mfaella>

Materiale didattico elaborato con i Proff. Sette e Bonatti

29 marzo 2022



Panoramica della lezione

Procedure come astrazioni

Propagazione dei data object

Bibliografia



Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di . . .

Ambiente di . . .

Ambiente di . . .

Esempio

Propagazione dei
data object

Bibliografia

Procedure come astrazioni



Procedure

Procedure come astrazioni

Procedure

Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.



Procedure

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.



Procedure

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.



Procedure

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

Procedure sono astrazioni di parti di programma in unità di esecuzione più piccole (invocazioni), in modo da nascondere i dettagli irrilevanti ai fini del loro (ri-)uso.

Vantaggi:

- Programmi più semplici da scrivere, leggere o modificare; suddivisione dei compiti in ogni brano di programma; progettazione top-down.
- Unità di programmi indipendenti o con dipendenze ben specificate a livello più alto.
- Riusabilità di brani di programmi; riduzione errori.



Astrazione procedurale

- Se si distinguono come unità di esecuzione, in ordine crescente di complessità, *espressioni*, *enunciati* (statement), *blocchi*, *programmi*, allora si definisce

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia



Astrazione procedurale

- Se si distinguono come unità di esecuzione, in ordine crescente di complessità, *espressioni*, *enunciati* (statement), *blocchi*, *programmi*, allora si definisce
- **astrazione procedurale** la rappresentazione di una unità di esecuzione attraverso un'altra unità più semplice (l'invocazione)

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione
Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia



Astrazione procedurale

- Se si distinguono come unità di esecuzione, in ordine crescente di complessità, *espressioni*, *enunciati* (statement), *blocchi*, *programmi*, allora si definisce
- **astrazione procedurale** la rappresentazione di una unità di esecuzione attraverso un'altra unità più semplice (l'invocazione)
- In pratica è la rappresentazione di un blocco attraverso un enunciato o una espressione.

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione
Invocazione di ...
Ambiente di ...
Ambiente di ...
Esempio

Propagazione dei
data object

Bibliografia



Dichiarazione di procedura

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

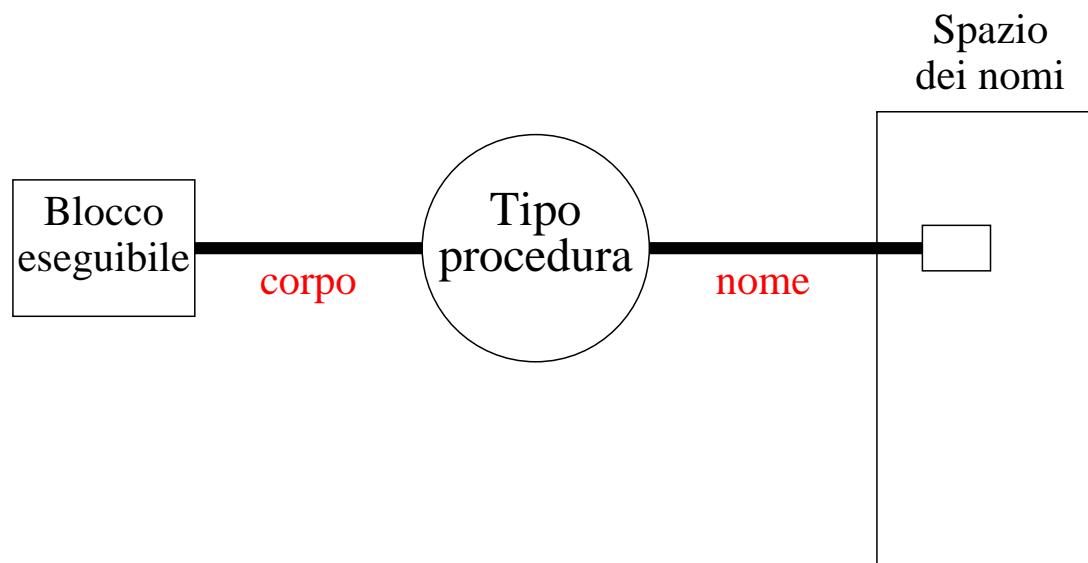
Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

- Causa la generazione di un legame tra il nome della procedura e il corpo della procedura (oltre a parametri, tipo di ritorno, etc.)
- Chiamiamo questo legame un oggetto “Tipo procedura”
- Il legame viene stabilito durante la compilazione





Invocazione di una procedura

Procedure come astrazioni

Procedure
Astrazione
procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

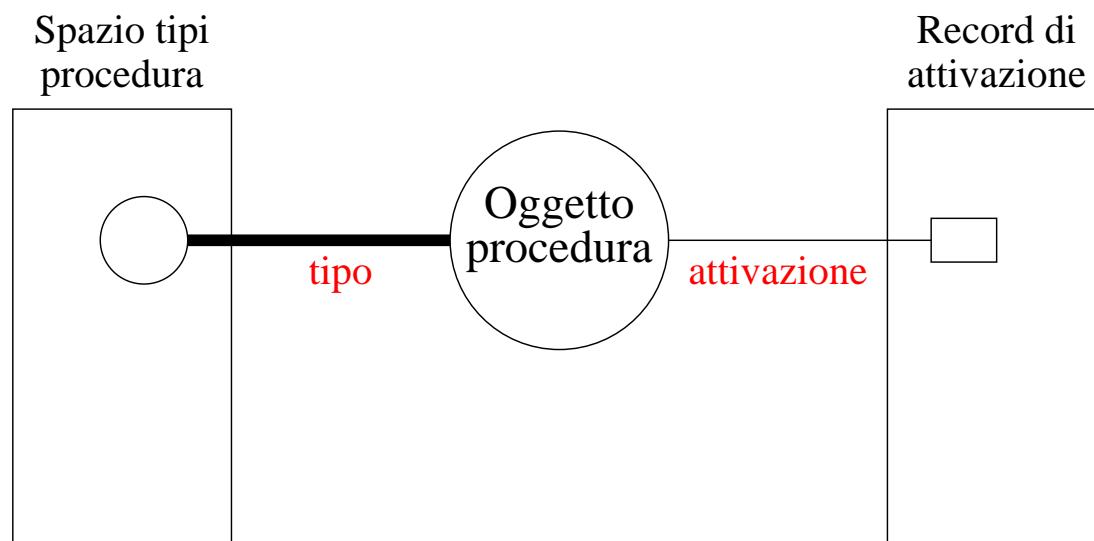
Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

- Causa la generazione di un legame tra un Tipo procedura e un Record di attivazione
- Chiamiamo questo legame un “Oggetto procedura”
- Il legame avviene durante l'esecuzione, nel momento in cui c'è l'invocazione della procedura
- Ogni invocazione diversa della stessa procedura causa la generazione di un nuovo “Oggetto procedura” con lo stesso legame di tipo, ma con diverso record di attivazione





Ambiente di esecuzione

Analogamente a quanto avveniva per un blocco in-line il **record di attivazione (RdA)** rappresenta l'intero ambiente di esecuzione di una procedura o funzione

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei data object

Bibliografia



Ambiente di esecuzione

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei data object

Bibliografia

Analogamente a quanto avveniva per un blocco in-line il **record di attivazione (RdA)** rappresenta l'intero ambiente di esecuzione di una procedura o funzione

A differenza del blocco anonimo, una procedura o funzione...

- Ha un nome e può essere invocata in vari punti del programma
- Può avere parametri
- Può avere un valore di ritorno (se funzione)

Quindi, l'RdA di una funzione contiene più informazioni di quello di un blocco in-line



Ambiente di esecuzione

Procedure come astrazioni

Procedure Astrazione
procedurale
Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei
data object

Bibliografia

L'RdA di una funzione contiene:

1. Puntatore di catena dinamica (link al prossimo RdA)
2. Puntatore di catena statica (new!) (opzionale, serve per scoping statico)
3. Indirizzo di ritorno (new!)
4. Indirizzo del risultato (new!)
5. Parametri (new!)
6. Ambiente locale (variabili locali e spazio per risultati intermedi)

L'ambiente non locale di una funzione viene recuperato tramite il puntatore di catena dinamica (scoping dinamico) o statica (scoping statico)



Esempio

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei data object

Bibliografia

Stack esecuzione

p: prima di chiamare s

p

```
program p;
var i;

procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
```



Esempio

Procedure come astrazioni

Procedure Astrazione

procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

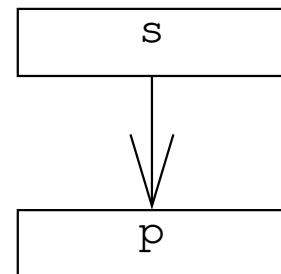
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

s: prima di chiamare r



program p;
var i;

procedure q;
begin
...
end;

procedure r;
begin
i:= i-1;
if i>0 then
r
else
q
end;

procedure s;
begin
r;
q
end;

begin
i:= 2;
s
end.



Esempio

Procedure come astrazioni

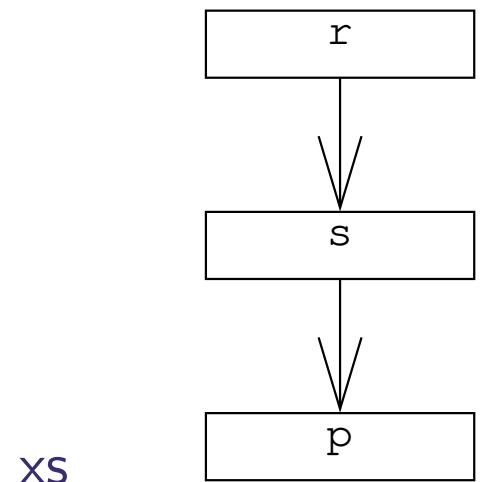
Procedure
Astrazione
procedurale
Dichiarazione
Invocazione di ...
Ambiente di ...
Ambiente di ...
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

r: prima di chiamare r



program p;
var i;

procedure q;
begin
...
end;

procedure r;
begin
i:= i-1;
if i>0 then
r
else
q
end;

procedure s;
begin
r;
q
end;

begin
i:= 2;
s
end.



Esempio

Procedure come astrazioni

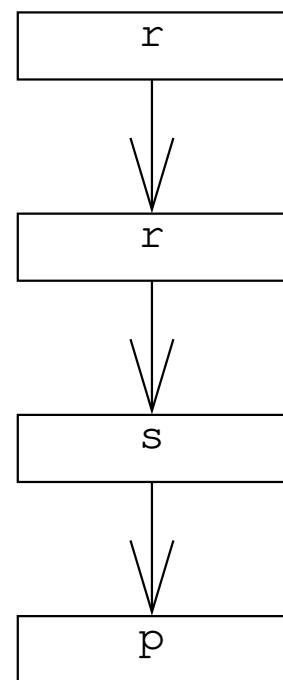
Procedure
Astrazione
procedurale
Dichiarazione
Invocazione di ...
Ambiente di ...
Ambiente di ...
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

r: prima di chiamare q



```
program p;
var i;

procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
```



Esempio

Procedure come astrazioni

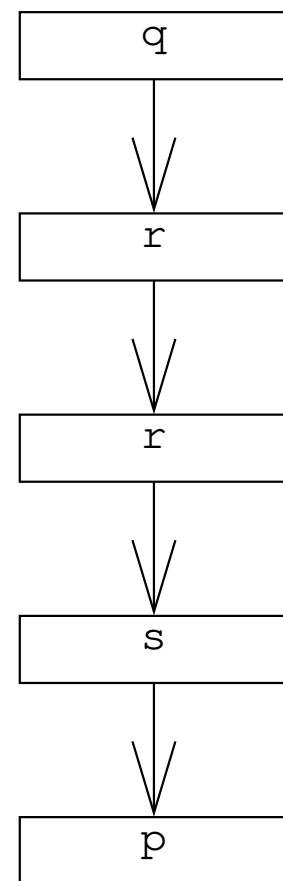
Procedure
Astrazione
procedurale
Dichiarazione
Invocazione di ...
Ambiente di ...
Ambiente di ...
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

q: prima di terminare



program p;
var i;

procedure q;
begin
...
end;

procedure r;
begin
i:= i-1;
if i>0 then
r
else
q
end;

procedure s;
begin
r;
q
end;

begin
i:= 2;
s
end.



Esempio

Procedure come astrazioni

Procedure Astrazione

procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

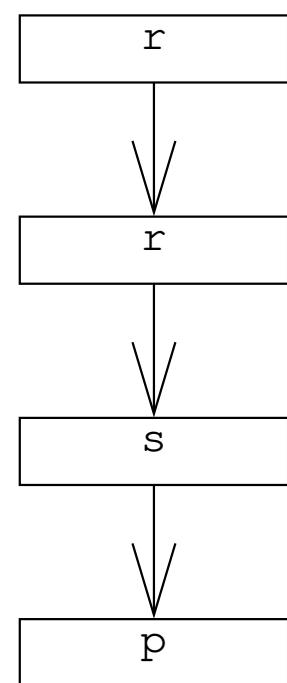
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

r: prima di terminare



```
program p;
var i;

procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
```



Esempio

Procedure come astrazioni

Procedure Astrazione

procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

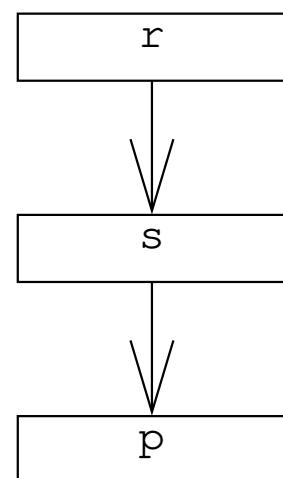
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

r: prima di terminare



```
program p;
var i;

procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
  end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
```



Esempio

Procedure come astrazioni

Procedure
Astrazione
procedurale
Dichiarazione
Invocazione di ...
Ambiente di ...
Ambiente di ...

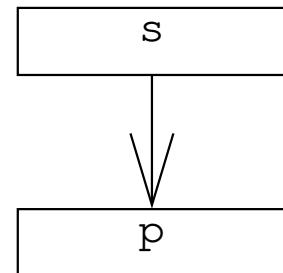
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

s: prima di q



program p;
var i;

procedure q;
begin
...
end;

procedure r;
begin
i:= i-1;
if i>0 then
r
else
q
end;

procedure s;
begin
r;
q
end;

begin
i:= 2;
s
end.



Esempio

Procedure come astrazioni

Procedure Astrazione

procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

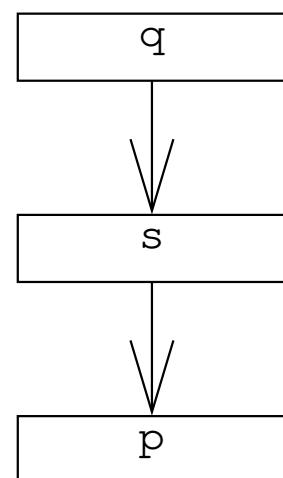
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

q: prima di terminare



```
program p;
var i;

procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
```



Esempio

Procedure come astrazioni

Procedure Astrazione

procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

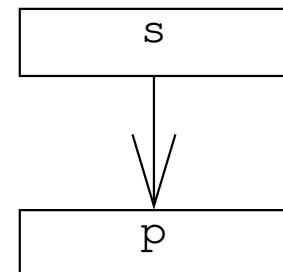
Esempio

Propagazione dei
data object

Bibliografia

Stack esecuzione

s: prima di terminare



program p;
var i;

procedure q;
begin
...
end;

procedure r;
begin
i:= i-1;
if i>0 then
r
else
q
end;

procedure s;
begin
r;
q
end;

begin
i:= 2;
s
end.



Esempio

Procedure come astrazioni

Procedure Astrazione procedurale

Dichiarazione

Invocazione di ...

Ambiente di ...

Ambiente di ...

Esempio

Propagazione dei data object

Bibliografia

Stack esecuzione

p: prima di terminare

p

```
program p;
var i;

procedure q;
begin
  ...
end;

procedure r;
begin
  i:= i-1;
  if i>0 then
    r
  else
    q
end;

procedure s;
begin
  r;
  q
end;

begin
  i:= 2;
  s
end.
```



Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C

Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia

Propagazione dei data object



Ambiente non locale

- L'*ambiente non locale* è l'insieme di tutti quei nomi (data object) a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio

[Procedure come astrazioni](#)

[Propagazione dei data object](#)

Ambiente non locale

[Implementazione](#)

[Il caso dei blocchi in-line](#)

[Il caso del C](#)

[Scoping statico con procedure annidate](#)

[Scoping statico \(2\)](#)

[Osservazioni su scoping dinamico](#)

[Esempio di scoping dinamico](#)

[Bibliografia](#)



Ambiente non locale

- L'*ambiente non locale* è l'insieme di tutti quei nomi (data object) a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio
- Tre tipologie di propagazione (scoping):
 1. Propagazione in ambito statico (scoping statico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente: propagazione di posizione.

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C
Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia



Ambiente non locale

- L'*ambiente non locale* è l'insieme di tutti quei nomi (data object) a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio
- Tre tipologie di propagazione (scoping):
 1. Propagazione in ambito statico (scoping statico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente: propagazione di posizione.
 2. Propagazione in ambito dinamico (scoping dinamico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura chiamante.

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C
Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia



Ambiente non locale

- L'*ambiente non locale* è l'insieme di tutti quei nomi (data object) a cui la funzione può riferirsi e che non sono dichiarati in quella funzione
- La provenienza di questi data object propagati dipende dal linguaggio
- Tre tipologie di propagazione (scoping):
 1. Propagazione in ambito statico (scoping statico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura che la contiene sintatticamente: propagazione di posizione.
 2. Propagazione in ambito dinamico (scoping dinamico). In questo caso l'ambiente non locale di una procedura è propagato dal programma o dalla procedura chiamante.
 3. Nessuna propagazione (in generale, l'uso di ambienti non locali è scoraggiato perché produce effetti collaterali non facilmente prevedibili)

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C
Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia



Implementazione

- Lo scoping viene realizzato inserendo nell'RdA un puntatore all'RdA della funzione da cui vengono propagati i data object



Implementazione

- Lo scoping viene realizzato inserendo nell'RdA un puntatore all'RdA della funzione da cui vengono propagati i data object
 - ◆ Scoping statico: viene usato il puntatore di catena statica
 - ◆ Scoping dinamico: viene usato il puntatore di catena dinamica
- Se viene richiesto l'accesso ad un dato che non è definito localmente, esso viene ricercato ricorsivamente seguendo la catena appropriata



Implementazione

- Lo scoping viene realizzato inserendo nell'RdA un puntatore all'RdA della funzione da cui vengono propagati i data object
 - ◆ Scoping statico: viene usato il puntatore di catena statica
 - ◆ Scoping dinamico: viene usato il puntatore di catena dinamica
- Se viene richiesto l'accesso ad un dato che non è definito localmente, esso viene ricercato ricorsivamente seguendo la catena appropriata



Il caso dei blocchi in-line

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C
Scoping statico con procedure annidate
Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia

L'RdA di un blocco in-line non contiene il puntatore di catena statica (anche in presenza di scoping statico) perché viene eseguito sempre nel contesto del blocco in cui è contenuto sintatticamente

Quindi, l'eventuale puntatore di catena statica *coinciderebbe con quello di catena dinamica*



Il caso del C

- Scoping statico
- Non supporta funzioni annidate
- Solo blocchi in-line annidati
- Lo scope non locale contiene: i blocchi esterni a quello corrente, poi la funzione corrente e infine lo scope globale
- Anche in questo caso, *non serve il puntatore di catena statica*

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C

Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia



Scoping statico con procedure annidate

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi
in-line

Il caso del C

Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia

Pseudo-codice:

```
program p;
  var a, b, c: integer;

  procedure q;
    var a, c: integer;
    procedure r;
      var a: integer;
      begin r          {variabili: a da r; b da p; c da q;
                           procedure: q ed s da p; r da q}
        ...
      end r;
      begin q          {variabili: a da q; b da p; c da q;
                           procedure: q ed s da p; r da q}
        ...
      end q;

    procedure s;
      var b: integer;
      begin s          {variabili: a da p; b da s; c da p;
                           procedure: q ed s da p}
        ...
      end s;

  begin p          {variabili: a, b, c da p;
                           procedure: q, s da p}
    ...
  end p.
```



Scoping statico (2)

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C

Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia

Supponendo una sequenza di attivazione (p, s, q, q, r), lo stack di esecuzione ha questa forma:



Scoping statico (2)

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C

Scoping statico con procedure annidate

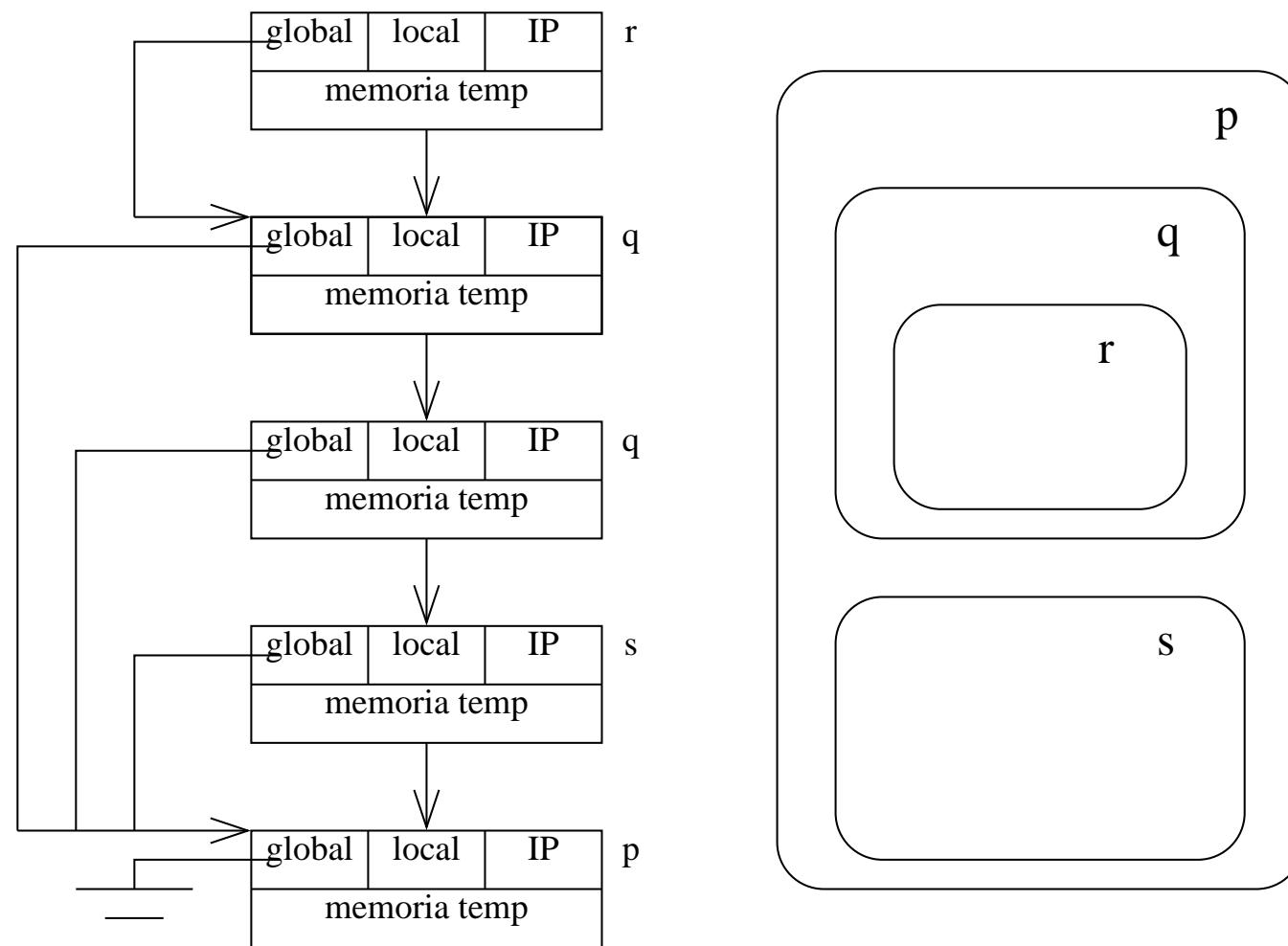
Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia

Supponendo una sequenza di attivazione (p, s, q, q, r), lo stack di esecuzione ha questa forma:





Osservazioni su scoping dinamico

Nello scoping dinamico:

- il puntatore di catena statica non è necessario
- è sufficiente il puntatore di catena dinamica



Osservazioni su scoping dinamico

Nello scoping dinamico:

- il puntatore di catena statica non è necessario
 - è sufficiente il puntatore di catena dinamica
-
- Vantaggio: è possibile trasmettere informazioni *indirettamente* a un'altra procedura (invece di doverle propagare con parametri)



Osservazioni su scoping dinamico

Nello scoping dinamico:

- il puntatore di catena statica non è necessario
 - è sufficiente il puntatore di catena dinamica
-
- Vantaggio: è possibile trasmettere informazioni *indirettamente* a un'altra procedura (invece di doverle propagare con parametri)
 - Svantaggio: è praticamente impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente.



Osservazioni su scoping dinamico

Nello scoping dinamico:

- il puntatore di catena statica non è necessario
 - è sufficiente il puntatore di catena dinamica
-
- Vantaggio: è possibile trasmettere informazioni *indirettamente* a un'altra procedura (invece di doverle propagare con parametri)
 - Svantaggio: è praticamente impossibile la determinazione dell'ambiente di esecuzione di una procedura durante la scrittura del codice sorgente.



Esempio di scoping dinamico

Procedure come astrazioni

Propagazione dei data object

Ambiente non locale

Implementazione

Il caso dei blocchi in-line

Il caso del C

Scoping statico con procedure annidate

Scoping statico (2)

Osservazioni su scoping dinamico

Esempio di scoping dinamico

Bibliografia

```
program p;
  var a: integer;
  procedure q;
    begin q           {vars: a da p o da r; procs: q, r da p}
      ...
    end q;
  procedure r;
    var a: integer;
    begin r          {vars: a da r; procs: q, r da p}
      ...
    end r;
begin p           {vars: a da p; procs: q, r da p}
  ...
end p.
```



Bibliografia

Procedure come astrazioni

Propagazione dei data object

Bibliografia

Bibliografia

Capitoli 7 (“La gestione della memoria”) e 9 (“Astrarre sul controllo”) di *Linguaggi di Programmazione, principi e paradigmi*, di Gabbrielli e Martini (2a edizione)

Richiami di C e C++

Il diagramma di *Memory Layout*

La notazione *Data Object*

Marco Faella
Università di Napoli Federico II

Richiami di C: dichiarazioni e definizioni

```
struct Person {  
    char name[100];  
    int age;  
};
```

```
struct Person p;
```

```
struct Person {  
    char name[100];  
    int age;  
} p;
```

Dichiarazione di un nuovo tipo di dati

Definizione di variabile di tipo Person, non inizializzata

Se globale, viene automaticamente inizializzata a zero (tutti zeri, anche i char)

Dichiarazione del tipo e **definizione** della variabile, congiunte

Inizializzazione di struct

```
struct Person {  
    char name[100];  
    int age;  
};
```

```
struct Person p1 =  
{ "Pippo", 30 };
```

Definizione e inizializzazione (vecchio stile)

```
struct Person p2 = {  
    .name = "Pippo",  
    .age = 30  
};
```

Definizione e inizializzazione (nuovo stile C99)

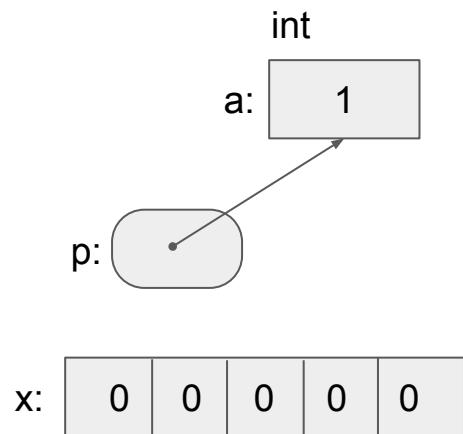
Diagramma di Memory Layout

- Mostra le allocazioni di memoria contigue
- Il valore di puntatori e riferimenti è rappresentato con frecce
- Non distingue tra stack e heap
- Versione grafica di *data object*

Codice C:

```
int a = 1
int *p = &a;
int x[5] = { 0, };
```

Memory layout:



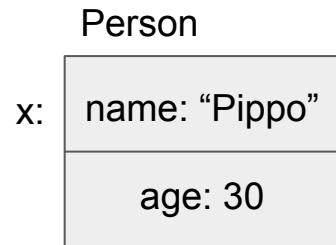
Memory Layout di struct (1)

Codice C:

```
struct Person {  
    char name[100];  
    int age;  
} x = {  
    .name = "Pippo",  
    .age = 30  
};
```

`sizeof(Person) → 104`

Memory layout:



Data Objects:

principale:

(loc1, x, {"Pippo", 30}, struct Person)

aggiuntivi:

(loc1, x.name, "Pippo", char[100])

(loc1+100, x.age, 30, int)

Memory Layout di struct (2)

Codice C:

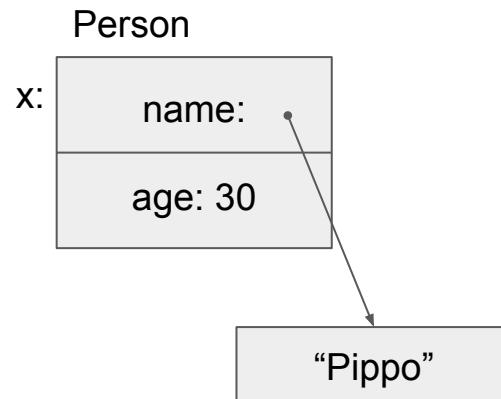
```
struct Person {  
    char *name;  
    int age;  
} x = {  
    .name = "Pippo",  
    .age = 30  
};
```

sizeof(Person) → 16

Perché 16?

alignment e padding

Memory layout:



Data Objects:

principali:

(loc1, x, {loc2, 30}, struct Person)
(loc2, anonimo, "Pippo", const char[])

aggiuntivi:

(loc1, x.name, loc2, char*)
(loc1+8, x.age, 30, int)

ipotizzando che `sizeof(char*)` sia 8

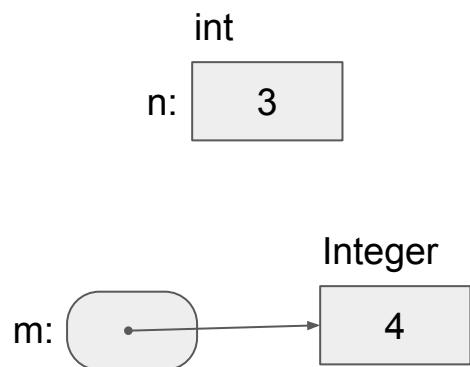
Esempio in Java: interi e Interi

Codice Java:

```
int n = 3;  
  
Integer m = 4;
```

Nota: grazie all'*autoboxing*, la seconda istruzione è equivalente a
`Integer m = Integer.valueOf(4)`

Memory layout:



Data Objects:

(loc1, n, 3, int)
(loc2, anonimo, 4, Integer)
(loc3, m, loc2, "riferimento a Integer")

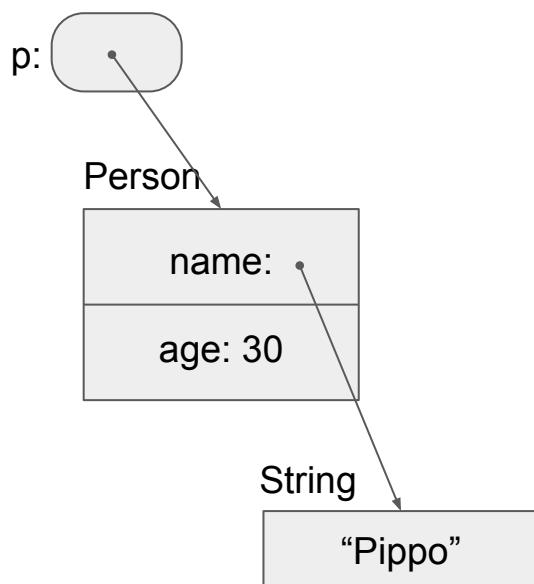
Esempio in Java: classi

Codice Java:

```
class Person {  
    String name;  
    int age;  
    ...  
}
```

```
Person p = new  
Person("Pippo", 30);
```

Memory layout:



Data Objects:

(loc1, anonimo, ..., Person)
(loc2, p, loc1, "riferimento a Person")
(loc3, anonimo, "Pippo", String)

Nota: questo memory layout è una semplificazione.
In realtà la stringa "Pippo" si trova in un array di
caratteri che è separato dall'oggetto di tipo String.

Tipi riferimento in Java

Codice Java:

```
Person p = new Person("Pippo", 30);
```

Indica il tipo
“riferimento a Person”

Indica il tipo Person

- Quando il nome di un tipo T (non primitivo) viene usato per creare un oggetto (cioè, new T(...)), crea un oggetto della classe T
- Quando il nome di un tipo T (non primitivo) viene usato per dichiarare una nuova variabile, indica il tipo “riferimento a T”

Richiami di C++: classi e allocazione automatica

```
class Person {  
private:  
    string name;  
    int age;  
public:  
    Person(string n, int a): name(n), age(a) {}  
};
```

The diagram shows the code for a `Person` class. A callout points to the parameter list `(string n, int a)` with the label "costruttore". Another callout points to the initialization part `: name(n), age(a)` with the label "sezione di inizializzazione". A third callout points to the empty body of the constructor `{ }` with the label "corpo vuoto".

Allocazione automatica di un oggetto `Person`:
(su stack se locale, su segmento statico se globale)

```
Person p = Person("Pippo", 30);
```

oppure:

```
Person p("Pippo", 30);
```

oppure:

```
Person p { "Pippo", 30 };
```

Richiami di C++: riferimenti

- Un riferimento è un alias, implementato come **puntatore costante**
- Sintatticamente, non si comporta come un puntatore, ma come l'oggetto puntato
- Uso tipico: per realizzare il passaggio di parametri per riferimento
 - Se il metodo non deve modificare l'argomento, si usa `const T&`
 - Se il metodo modifica l'argomento, si usa `T&`

Con puntatore:

```
void f(struct Big *big) {  
    cout << big->field;  
    cout << big;   ◯ Indirizzo della struct  
    cout << *big; ◯ Contenuto della struct  
    big->field = 0;  
    big = nullptr;  
}
```

Con riferimento:

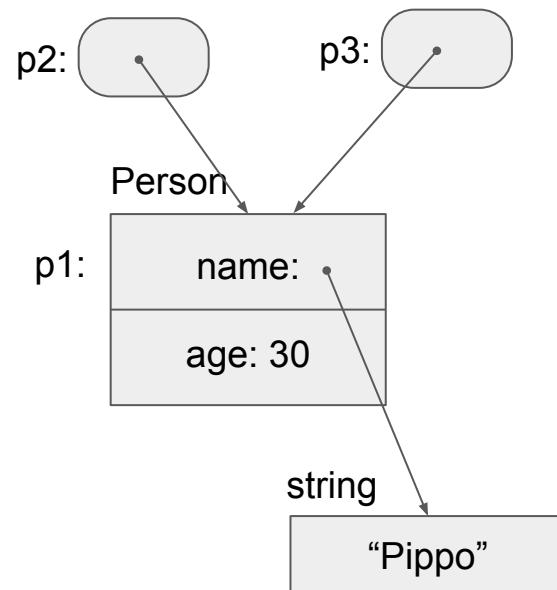
```
void f(struct Big &big) {  
    cout << big.field;  
    cout << big;   ◯ Contenuto della struct  
    cout << &big;  ◯ Indirizzo della struct  
    big.field = 0;  
    big = nullptr; ◯ Errore di comp.  
}
```

Esempio in C++:

Codice C++:

```
class Person {  
    string name;  
    int age;  
    ...  
};  
  
Person p1 =  
Person("Pippo", 30);  
  
Person &p2 = p1;  
Person *p3 = &p1;
```

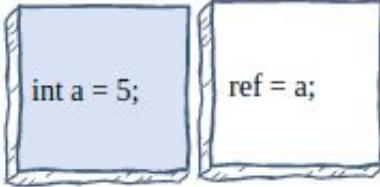
Memory layout:



Data Objects:

(loc1, p1, ..., Person)
(loc2, p2, loc1, riferimento a Person)
(loc3, p3, loc1, puntatore a Person)
(loc4, anonimo, "Pippo", string)

Attenzione! (al materiale che si trova online)



Creating a reference to **a** just makes an alias for it;
it does not "point" to **a** by storing its address in a
separate memory location

Da:

[https://www.educative.io/edpresso/
differences-between-pointers-and-references-in-cpp](https://www.educative.io/edpresso/differences-between-pointers-and-references-in-cpp)

Errato!

I riferimenti si comportano come “alias”, ma sono implementati come puntatori costanti

Dimostrazione pratica:

```
class Person { ... }
struct PersonRef {
    Person &p;
}
```

```
sizeof(Person) → 40
sizeof(PersonRef) → 8
```

Data Model a confronto

Java

Le variabili non primitive contengono riferimenti

Le variabili primitive contengono valori

C#

Classi e array si manipolano per riferimento

I tipi primitivi e le struct per valore

C++

Per qualsiasi tipo, si può scegliere tra valore, puntatore e riferimento

Type equivalence

+

evoluzione dei sistemi di tipi

+

**classificazione dei linguaggi di
programmazione**

Ritorno al passato

All'inizio esistevano solo i tipi elementari

E nessuna gerarchia di classi...

Poi sono stati introdotti i tipi user-defined

Ancora niente classi

Inizialmente semplici ridenominazioni di tipi elementari oppure nomi di record

Un assegnamento $x = y$ (o $x := y$) o un passaggio di parametri quando era valido?

Lo stabilisce la nozione di *type equivalence* adottata dal linguaggio

Forme di equivalenza

Name equivalence

I tipi di x e y devono avere lo stesso nome
cioè essere lo stesso tipo

Structural equivalence

I tipi di x e y devono avere la stessa rappresentazione interna
Apparentemente più snello e flessibile, in realtà aumenta la
possibilità di errori

```
Euro x;  
Dollar y;  
z = x+y ; // che senso ha?
```

Esempi

Pascal: name equivalence

C (e C++): entrambe!

Quasi sempre structural tranne che per le struct

```
typedef int money;
typedef int apples;

typedef struct{ int a; } S1;
typedef struct{ int a; } S2;

int main(){
    money x=0;
    apples y=0;
    int z = x+y; // non fa una piega
    S1 a;
    S2 b;
    a = b; // questo invece non lo compila
```

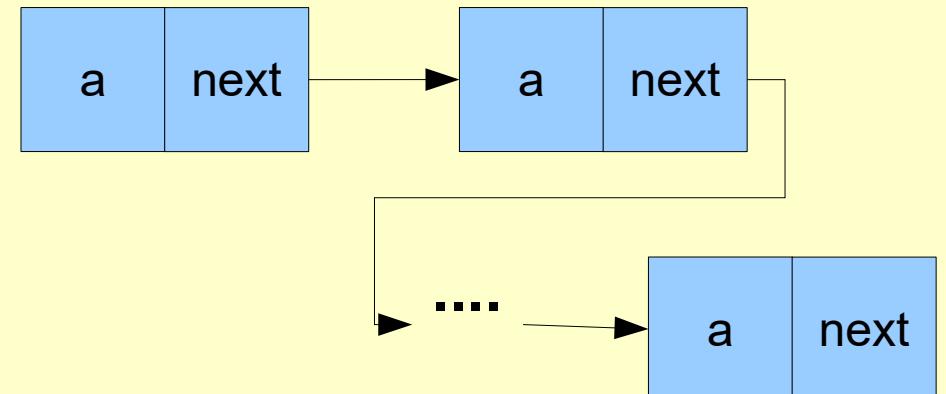
Ecco perchè

Verificare se due struct sono strutturalmente equivalenti richiede di verificare una proprietà chiamata *bisimulazione*

Caso semplice: sia S1 che S2 generano tutte le catene come quella a destra

```
struct S1 { int a; struct S1* next; };

struct S2 { int a; struct S3* next; };
struct S3 { int a; struct S2* next; };
```



Complicazioni in presenza di record varianti

Compatibilità di tipi

Con l'avvento dei linguaggi a oggetti l'equivalenza viene rimpiazzata da *compatibilità*

Qualunque sottotipo di T è compatibile con T

Nei linguaggi O.O. più comuni la compatibilità è basata su *nome* piuttosto che *struttura*

Due classi con nomi diversi sono diverse

Anche se hanno gli stessi attributi e gli stessi metodi

Inoltre una classe per essere sottotipo di un'altra deve essere *esplicitamente* dichiarata tale (keyword *extends*)

La struttura non conta neanche in questo caso

Evoluzione dei sistemi di tipi

Tipi di dato elementari

Tipi user-defined (ad es. Pascal, C)

Solo strutture dati

Interfacce e tipi di dato astratti (ad es. Modula, Ada)

Encapsulation: strutture dati accessibili solo attraverso specifiche procedure

Modificatori di accesso

Disaccoppiamento interfaccia/implementazione

Gerarchie di tipi

Ed ecco finalmente i linguaggi O.O.

Caratteristiche utili alla classificazione dei linguaggi

Paradigma di riferimento

imperativo, O.O, funzionale, logico

Scoping (statico/dinamico)

Gestione della memoria

Allocazione dinamica, garbage collection/esplícita, ...

Sistema di tipi

strong/weak, encapsulation, equivalence/compatibility,
polimorfismo (di 4 tipi), *type inference*...

Supporto alle eccezioni

Eventualmente integrato con type checking... vedi ML

Supporto al parallelismo

Memoria condivisa (*synchronized*), scambio di messaggi (nel senso *RMI*), gestione del nondeterminismo, fairness

Caratteristiche utili alla classificazione dei linguaggi

Naturalmente determinano come si usa al meglio un dato linguaggio

E quali sono gli errori da evitare

Ad es. sapendo cosa il compilatore può o non può fare per voi