



LA AST PER JAVA



Struttura

- ✦ Java AST: tre componenti fondamentali
 - **Workspace** per la memorizzazione dei programmi in esecuzione
 - **Stack** per la gestione dei binding
 - **Heap** per la gestione della memoria dinamica
- ✦ Oltre a questi componenti la AST è caratterizzata da uno spazio di memoria dinamica che serve per memorizzare le tabelle dei metodi degli oggetti
 - Per semplicità ora non consideriamo questa parte

Cosa cambia rispetto a Ocaml?



- ☛ Ogni nome presente sullo stack denota una entita' mutabile (oggetti, array, ...)
- ☛ Heap contiene solamente oggetti e array
- ☛ Java prevede il valore speciale null, per indicare che non esiste alcun oggetto associato al nome
- ☛ Le tabelle dei metodi
- ☛ La gestione dell'ereditarieta'.

Allocazione di oggetti sullo heap



```
class Node {
  private int elt;
  private Node next;
}
```

```
:
Node n = new Node(1, null)
```

HEAP	
Node	
elt	1
next	null

Le variabili di istanza possono essere mutabili o meno

Nota importante: a run time sono presenti informazioni di tipo esemplificare dalla "annotazione di tipo" **Node** memorizzato nello heap. Il perche' lo capirete in seguito!!!

Allocazione di array sullo heap



Int [] a = [7, 3, 5, 0]

HEAP			
Int[]			
length		4	
7	3	5	0

Il valore di length e' fissato
Gli elementi dell'array sono mutabili

Nota importante: a run time sono presenti informazioni di tipo
 esemplificare dal "tipo" array di interi e la dimensione (length)
 memorizzato nello heap.

```
class Node {
  private int elt;
  private Node next;

  public Node (int e0, Node n0) {
    elt = e0;
    next = n0;
  }

  public int m () {
    Node n1 = new Node (1,null);
    Node n2 = new Node (2, n1);
    Node n3 = n2;
    n3.next.next = n2;
    Node n4 = new Node (4, n1.next);
    n2.next.elt = 17;
    return n1.elt;
  }
}
```

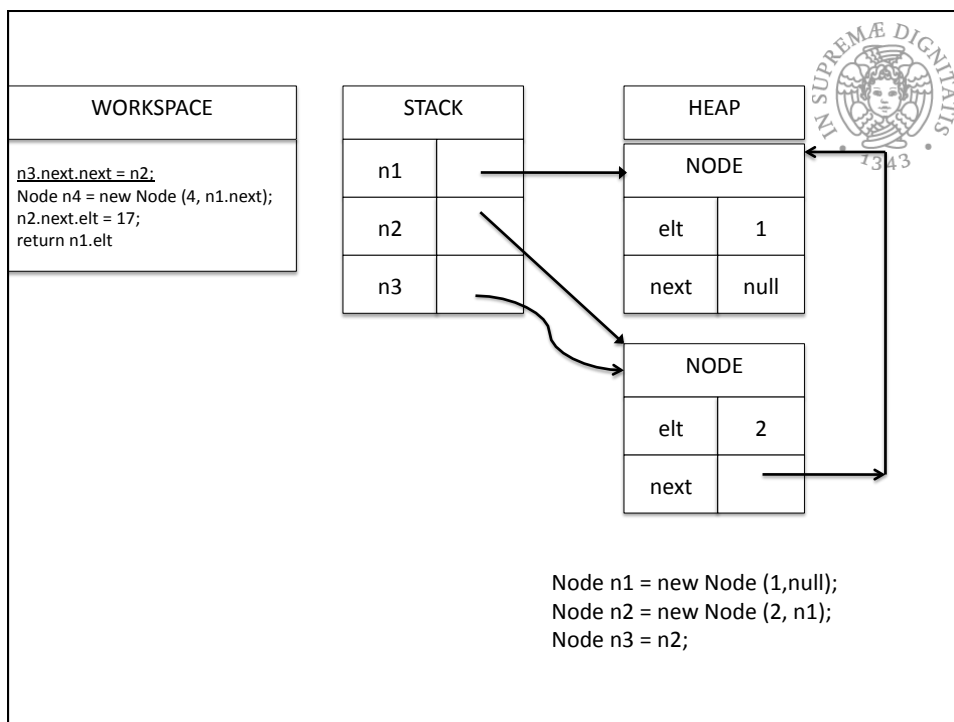


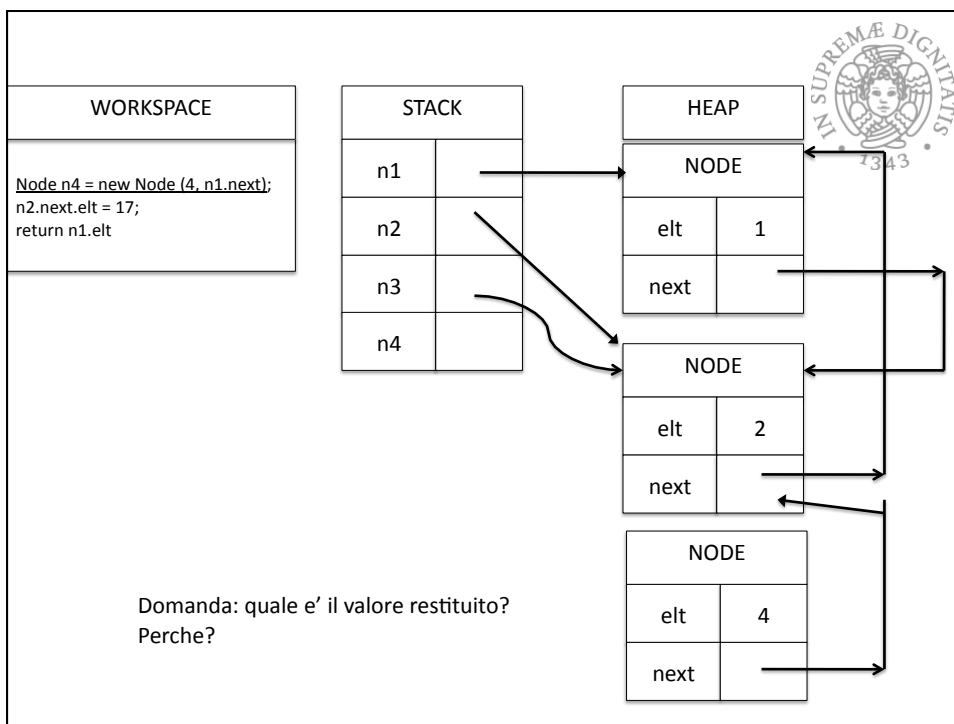
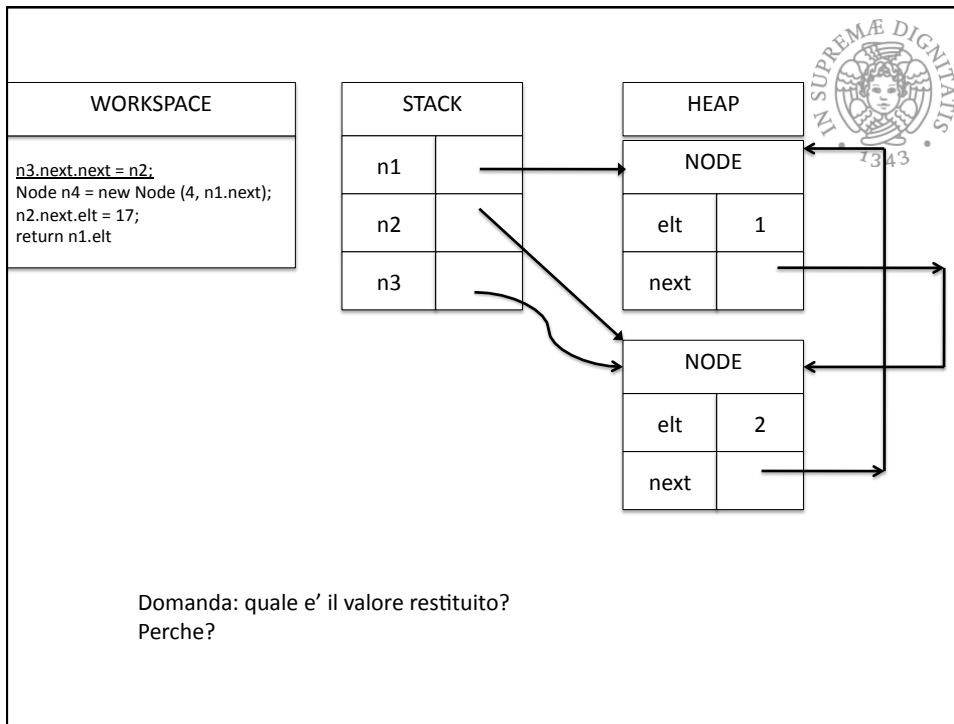
E
S
E
M
P
I
O

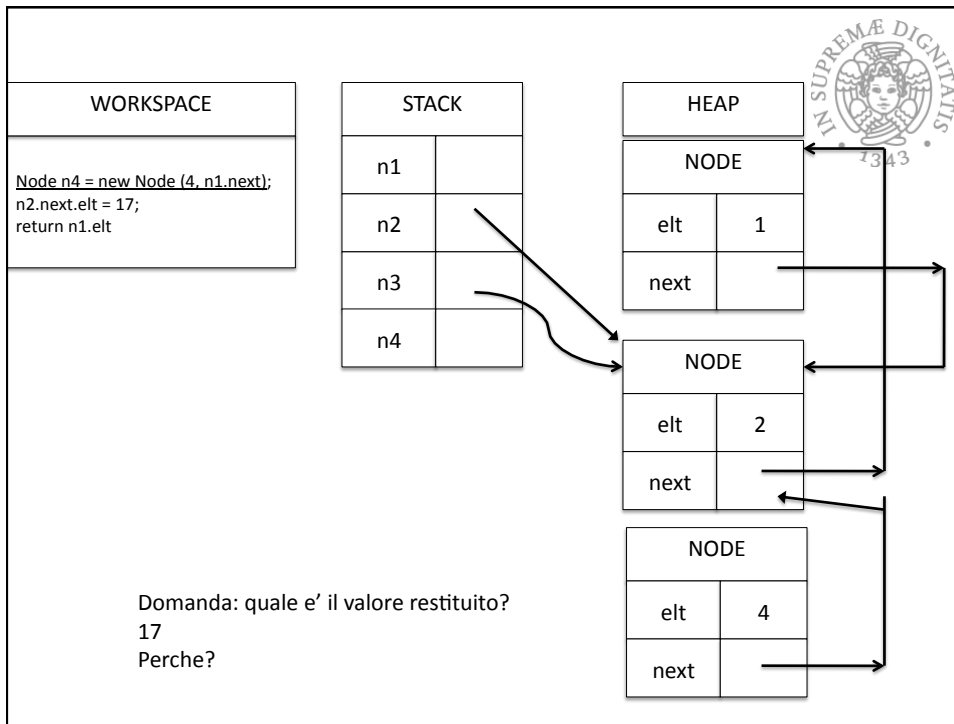
Esempio di AST



- ✎ Supponiamo di valore valutare l'invocazione
 - Node.m();
- ✎ La prima cosa che si deve osservare e' che l'invocazione del metodo restituisce un valore intero (che non e' un oggetto).
- ✎ Lo stack deve contenere lo spazio per poter memorizzare il valore restituito dall'invocazione del metodo
 - Intuitivamente una variable ans di tipo int.







EREDITARIETA'

Ereditarieta' di Classi



- ✎ Esamineremo solamente la nozione di ereditarieta' tra classi.
- ✎ La nozione di ereditarieta' vale anche per le interface: dettagli nelle note didattiche.
- ✎ Strumento tipico dell'OOP per riusare il codice e creare una gerarchia di astrazioni
- ✎ Generalizzazione: Una superclasse generalizza una sottoclasse fornendo un comportamento condiviso dalle sottoclassi
- ✎ Specializzazione: Una sottoclasse specializza (concretizza) il comportamento di una super-classe

Perche' e' importante



- ✎ Permette di specializzare il comportamento di una classe, prevedendo nuove funzionalità, mantenendo le vecchie e quindi senza influenzare codice cliente già scritto.
- ✎ Reuso del codice!!
- ✎ Subtype polymorphism: tramite l'ereditarieta' un variabile puo' assumere tipi (di classi) differenti.
- ✎ Una funzione con parametro formale di tipo T puo' operare con un parametro attuale di tipo T' a patto che T' sia un sottotipo di T.

Subtyping



- ✦ B e' un sottotipo di A: "every object that satisfies interface B also satisfies interface A"
- ✦ Obiettivo metodologico: codice scritto guardando la specifica di A opera correttamente anche se viene usata la specifica di B

15

Sottotipi e principio di sostituzione



- ✦ B e' un sottotipo di A: B puo' essere sostituito per A.
 - Una istanza del sottotipo soddisfa le proprieta' del supertipo.
 - Una istanza del sottotipo puo' avere maggiori vincoli di quella del supertipo
- ✦ Questo non e' sempre vero in Java

16



- ✚ Sottotipo e' una nozione semantica:
- ✚ B e' un sottotipo di A <-> un oggetto di B si puo' mascherare come un oggetto di A in tutti i possibili contesti
- ✚ Ereditarieta' e' una nozione di implementazione
 - Creare una nuova classe evidenziando solo le differenze (il codice nuovo)
- ✚ Ora esamineremo gli aspetti concreti di ereditarieta' in Java: in seguito vedremo la parte semantica.

17

Ereditarieta' in Java



- ✚ Una sottoclasse si definisce usando la parola chiave extends:
 - **class B extends A { . . }**
- ✚ Osservazione: l'ereditarieta' in Java è semplice:
 - Una classe può implementare più interfacce, ma estendere solo una superclasse
 - Questo non vale in altri linguaggi a oggetti: esempio classico C++

Esempio



```
class D {
  private int x;
  private int y;
  public int addBoth() { return x + y; }
}
```

La classe C eredita implicitamente i campi definiti dalla classe D

```
class C extends D { // C e' un sottotipo di D
  private int z;
  public int addThree() {return (addBoth() + z); }
}
```

Nella classe D non sono visibili le variabili e metodi dichiarati **private**. Quindi fanno parte dello stato degli oggetti istanziati e non sono riferibili (direttamente) dal nuovo metodo

Esempio



```
class D {
  protected int x;
  protected int y;
  public int addBoth() { return x + y; }
}
```

La classe C eredita implicitamente i campi definiti dalla classe D

```
class C extends D { // C e' un sottotipo di D
  private int z;
  public int addThree() {return x + y + z; }
}
```

Nella classe D sono visibili le variabili e metodi dichiarati **protected**. Quindi sono riferibili direttamente.

Metodo Costruttore e Super



- ✎ Un aspetto critico della nozione di ereditarietà è che il metodo costruttore non è ereditato.
- ✎ Tipicamente il metodo costruttore della sottoclasse deve accedere alle variabili di istanza, anche della superclasse.
- ✎ Java fornisce un meccanismo specifico per affrontare questo aspetto.

```
class D {
    private int x;
    private int y;
    public D (int initX, int initY) {
        x = initX; y = initY;
    }
    public int addBoth() { return x + y; }
}

class C extends D { // C e' un sottotipo di D
    private int z;
    public C (int initX, int initY, int initZ) {
        super(initX, initY); // invocazione del costruttore di D
        z = initZ;
    }

    public int addThree() {return (addBoth() + z); }
}
```



this



- ✎ All'interno di un metodo o di un costruttore, la parola chiave **this** permette di riferire l'oggetto corrente.
- ✎ **this** e' un riferimento all'oggetto corrente - l'oggetto di cui metodo o il costruttore viene chiamato.

This (esempio)



```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

This (costruttore implicito)



```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 0, 0);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

Chiamata al costruttore con quattro parametri

Esempio



```
public class Point {
    protected final int x, y;
    private final String name;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
        name = makeName();
    }

    protected String makeName() {
        return "["+x+", "+y+"]";
    }

    public final String toString() {
        return name;
    }
}

public class ColorPoint extends Point{
    private final String color;

    public ColorPoint(int x, int y, String color){
        super(x,y);
        this.color = color;
    }

    protected String makeName() {
        return super.makeName() + ":" + color;
    }

    public static void main(String [] args) {
        System.out.println(new ColorPoint(4, 2,
            "viola"));
    }
}
```

Cosa succede?

Esempio



```

public class Point {
    protected final int x, y;
    private final String name;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
        name = makeName();
    }

    protected String makeName() {
        return "["+x+", "+y+"]";
    }

    public final String toString() {
        return name;
    }
}

public class ColorPoint extends Point{
    private final String color;

    public ColorPoint(int x, int y, String color){
        super(x,y);
        this.color = color;
    }

    protected String makeName() {
        return super.makeName() + ":" + color;
    }

    public static void main(String [] args) {
        System.out.println(new ColorPoint(4, 2,
            "viola"));
    }
}

```

[4, 2]:null

Upcasting&downcasting



- Supponiamo che T sia una sottoclasse di S (in una gerarchia)
- **Upcasting**: un oggetto di tipo T puo' essere legato a una variabile di tipo S
- **Downcasting**: un oggetto di tipo S puo' essere legato a una variabile di tipo T.

```

class Vehicle { ...};
class Car extends Vehicle; // Car sottotipo di Vehicle

Vehicle v =(Vehicle) new Car(); // upcasting
Car c = (Car)new Vehicle(); // downcasting

```

Upcasting&downcasting



- ✿ Upcasting e' implicito
- ✿ Downcasting deve essere esplicito
 - Non sono possibili operazioni di cast al di fuori della struttura descritta dalla gerarchia!!

Metodi addizionali



- ✿ esistono vari metodi definiti nella classe Object che possono essere ereditati quando ha senso o ridefiniti da qualunque classe
- ✿ alcuni esempi
 - equals
 - clone
 - toString

equals



- ✎ in Object il metodo equals verifica se due oggetti sono lo stesso oggetto
 - non se i due oggetti hanno lo stesso stato
 - Deve essere ridefinita per i tipi non modificabili
 - ✓ in termini di uguaglianza fra gli stati
- ✎ in Object c'è anche un metodo hashCode che produce, dato un oggetto, un valore da usare come chiave in una tabella Hash
 - stesso valore per oggetti equivalenti (secondo equals)
 - se un tipo non modificabile è usato come chiave, deve ridefinire anche hashCode

31

clone



- ✎ in Object genera una copia dell'oggetto
 - nuovo oggetto con lo stesso stato
- ✎ questa implementazione non è sempre corretta
 - creando una situazione di condivisione (con trasmissione di modifiche) non desiderata
- ✎ il metodo viene ereditato solo se l'header della classe contiene la clausola implements Cloneable
- ✎ se non va bene quella di default si deve reimplementare

32

toString



- ✎ in Object genera una stringa contenente il tipo dell'oggetto ed il suo Hash code
- ✎ normalmente si vorrebbe ottenere una stringa composta da
 - tipo
 - valori dello stato
- ✎ se se ne ha bisogno, va ridefinita sempre

33



TIPI STATICI E DINAMICI

Tipi



✎ Il tipo **statico** di una variabile e' il tipo della classe (o della interfaccia) che definisce quali oggetti possono essere legati a quella variabile.

✎ Esempio

```
public class C { .....}
```

```
C c = new C();
```

C e' il tipo statico della variabile c.

Tipi statici e dinamici



- Il tipo **statico** di una espressione e' il tipo che descrive il valore calcolato dall'espressione solamente in base alla struttura testuale della espressione (senza valutarla).
- Il tipo **dinamico** di un oggetto e' il tipo della classe di cui l'oggetto e' istanza.

Statico vs Dinamico



- ✦ Nel caso di Ocaml la differenza tra tipi statici e tipi dinamici non aveva una utilita' esplicita.
- ✦ La presenza della nozione di ereditarieta' fa emergere chiaramente questa nozione.
- ✦ Il tipo dinamico di una variabile o di una espressione e' sempre un sottotipo del tipo statico.

Tipi e Gerarchia



```
public class Shape { ... }
```

```
public class point extends Shape { ... }
```

```
public class Circle extends Shape { ... }
```

```
Point p = new Point ()  
Circle c = new Circle ();  
Shapes1=p; // Linea A  
Shapes2=c; // Linea B  
s2 =p; // Linea C
```

Esempio



```
public class Shape { ... }

public class point extends Shape { ... }

public class Circle extends Shape { ... }
```

```
Point p = new Point ()
Circle c = new Circle ();
Shape s1=p; // Linea A
Shape s2=c; // Linea B
s2 =p; // Linea C
```

Quale e' il tipo statico di s1 alla linea A?

Quale e' il tipo dinamico di s1 alla linea A dopo l'assegnamento?

Esempio



```
public class Shape { ... }

public class point extends Shape { ... }

public class Circle extends Shape { ... }
```

```
Point p = new Point ()
Circle c = new Circle ();
Shape s1=p; // Linea A
Shape s2=c; // Linea B
s2 =p; // Linea C
```

Quale e' il tipo statico di s1 alla linea A?
Shape

Quale e' il tipo dinamico di s1 alla linea A dopo l'assegnamento?
Point

Esempio



```
public class Shape { ... }

public class point extends Shape { ... }

public class Circle extends Shape { ... }
```

```
Point p = new Point ()
Circle c = new Circle ();
Shape s1=p; // Linea A
Shape s2=c; // Linea B
s2 =p; // Linea C
```

Quali sono i tipi dinamici di s2?

Circle alla Linea B
Point alla Linea C

Esempio



```
public class Shape { ... }

public class point extends Shape { ... }

public class Circle extends Shape { ... }

public Shape asShape (Shape s) { return s; }
```

```
Point p = new Point ()
Circle c = new Circle ();
Shape s1=p; // Linea A
Shape s2=c; // Linea B
s2 =p; // Linea C
```

Quale e' il tipo statico di asShape(p)?
Shape

Quale e' il tipo dinamico di asShape(p)?
Point