PROGRAMMAZIONE 2
4. Un modello operazionale per Java

Liste in Java (*à la* OCaML)

```
class Cons implements StringList {
    private String head;
    private StringList tail;
  public Cons (String h, StringList t) {
        head = h; tail = t; }
    public boolean isNil( ) {
        return false; }
    public String hd( ) {
        return head ; }
    public StringList tl( ) {
          return tail; }
}
```

```
interface StringList {
    public boolean isNil( );
    public String hd( );
    public StringList tl( );
}
```

```
class Nil implements   StringList {
    public boolean isNil( ) {
        return true; }
    public String hd( ) {
        return null; }
    public StringList tl( ) {
        return null; }
}
```

Operare su liste

```
StringList x = new Cons(”Fra", new Cons(“Fra", new Nil( )))
```

- Regole pragmatiche generali
  - Per ogni tipo di dato definire la relativa interfaccia
  - Aggiungere una classe per ogni costruttore

Operare su liste

- Con ricorsione…

```
public static int numberOfStrings (StringList pl) {
    if (pl.isNil( )) { return 0; }
    else { return 1 + numberOfStrings (pl.tl( )); }
}
```

- …o iterazione!

```
public static int numberOfStrings (StringList pl) {
    int count = 0 ;
    StringList curr = pl;
    while (! Curr.isNil( )) {
        count = count + 1;
        curr = curr.tl( );
    }
    return count ;
}
```

Le stringhe in Java

Java String

- Le stringhe (sequenze di caratteri) in Java sono una classe predefinita
  - **"3" + " " + "Volte 3"** equivale a "**3 Volte 3"**
  - Il "+" è anche l'operatore di concatenazione di stringhe

- Le stringhe sono oggetti immutabili (*a là* OCaml)

Uguaglianza

- Java ha due operatori per testare l'uguaglianza
  - o1 == o2 restituisce true se le variabili o1 e o2 denotano lo stesso riferimento (pointer equality)
  - o1.equals(o2) restituisce true se le variabili o1 e o2 denotano due oggetti identici (deep equality)
- Esempio
  - String("test").equals("test") --> true
  - new String("test") == "test" --> false
  - new String("test") == new String("test") --> false

Un quesito interessante…

```
String s1 = "Java";
String s2 = "Java";
```

```
s1.equals(s2)  // true… perché?
s1==s2  // true… perché?
```

Un quesito più standard…

```
String str1 = new String("Java");
String str2 = new String("Java");

str1.equals(str2)  // true: stesso contenuto
str1==str2  // false: oggetti differenti
```

Abstract Stack Machine

- **Abstract Stack Machine**: modello computazionale per Java che permette di descrivere la nozione di stato modificabile
- Modello astratto: nella seconda parte del corso esamineremo nel dettaglio gli aspetti relativi alla realizzazione dei linguaggi di programmazione

Struttura

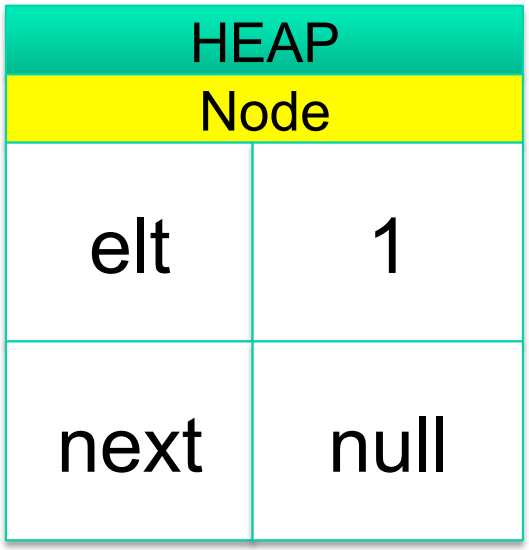
- **Java ASM**: tre componenti fondamentali
  - **Workspace** per la memorizzazione dei programmi in esecuzione
  - **Stack** per la gestione dei binding
  - **Heap** per la gestione della memoria dinamica
- Oltre a questi componenti la **ASM** è caratterizzata da uno spazio di memoria dinamica che serve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

ASM

- Buon modello per comprendere come funzionano i programmi Java
- Definisce in modo chiaro come sono gestite le strutture dati
- Permette di trattare in modo omogeneo la gestione degli oggetti
- Visione **semplificata** della macchina astratta per la realizzazione del linguaggio

Allocazione di oggetti sullo heap

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

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



Le variabili di istanza possono essere mutabili o meno

Nota: a run-time sono presenti informazioni di tipo esemplificate dalla “annotazione di tipo” **Node** memorizzate nello heap. Il perché si capirà in seguito!!!

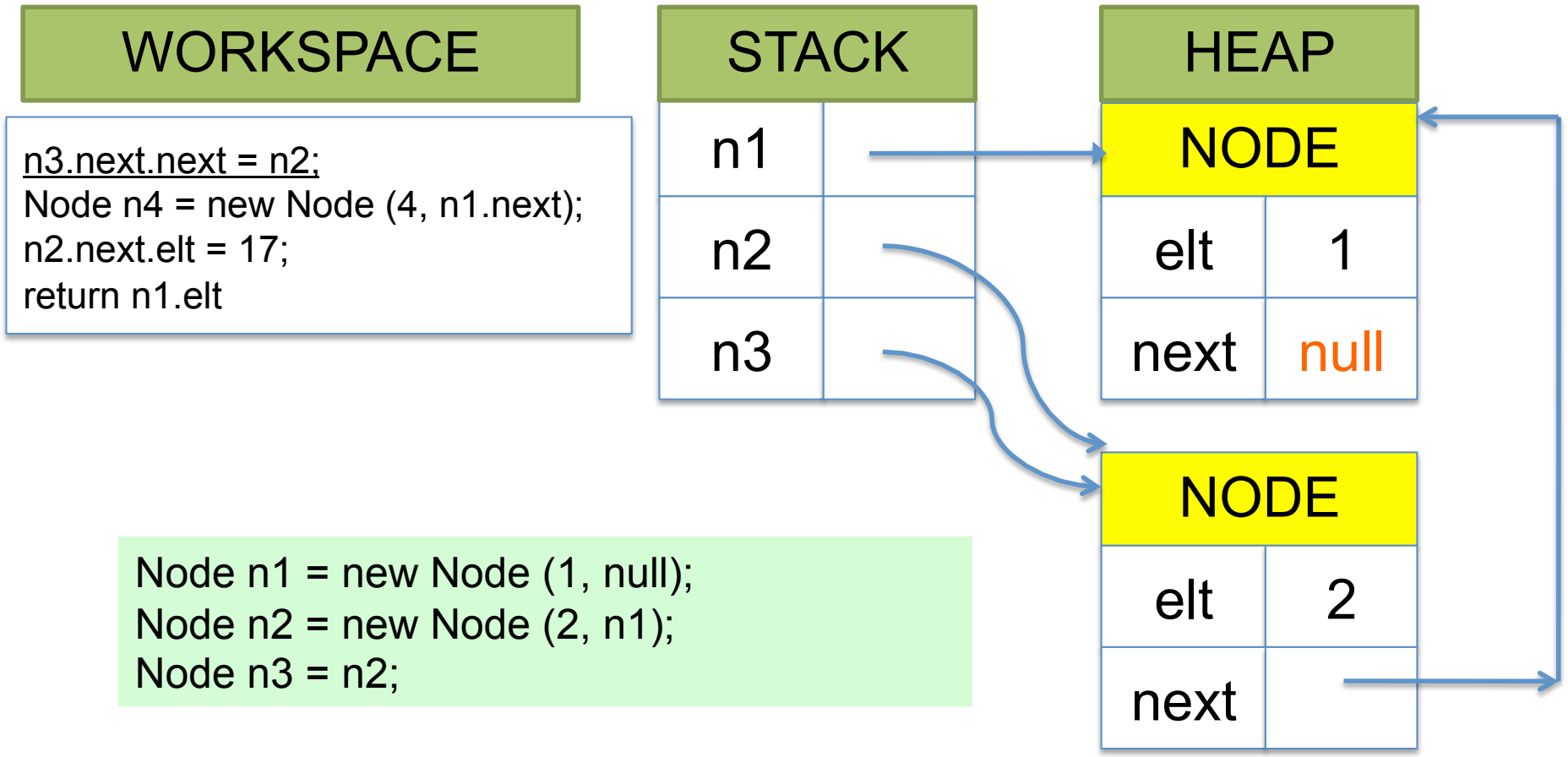
Esempio

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

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

```
public static 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;
}
```

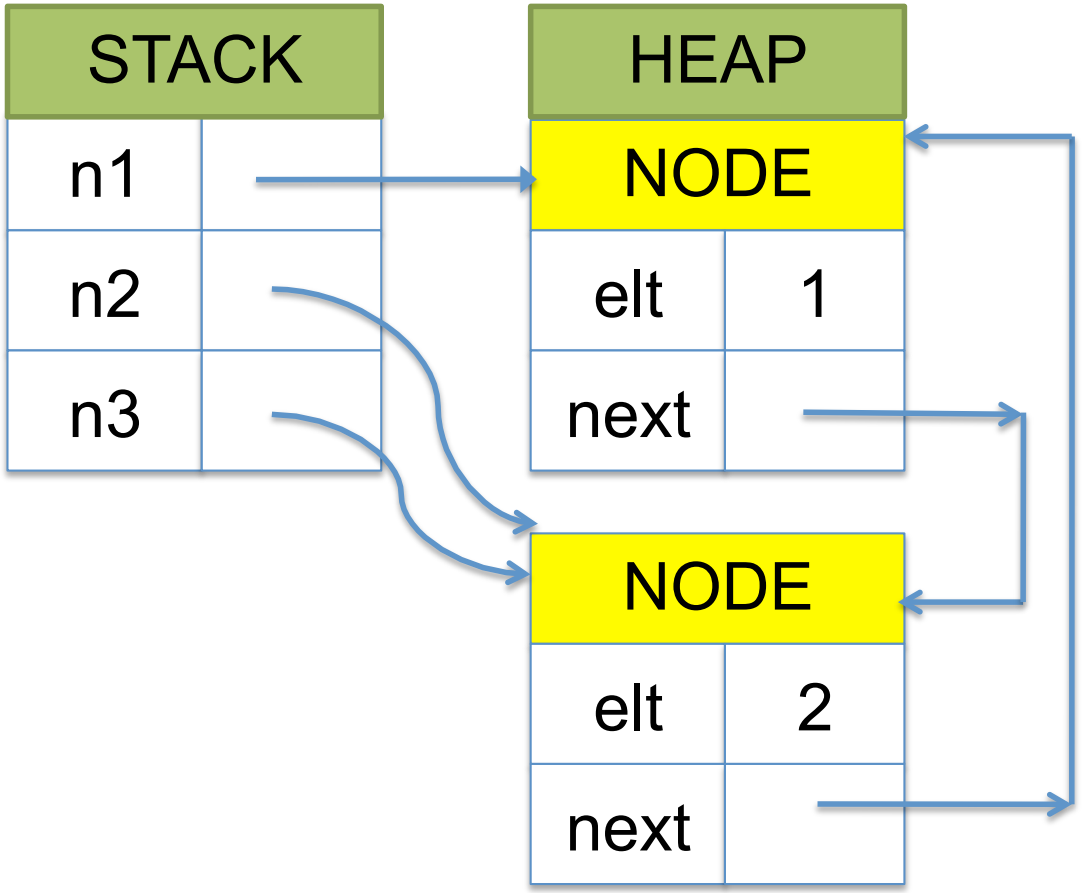
Esempio di ASM

- Supponiamo di voler valutare l'invocazione
  - Node.m( );
- La prima cosa da osservare è che l'invocazione del metodo statico restituisce un valore intero (che non è un oggetto)
- Lo stack deve contenere lo spazio per memorizzare il valore restituito dall'invocazione del metodo
  - intuitivamente una variable di tipo int
  - per semplicità lo omettiamo


WORKSPACE
n3.next.next = n2;
Node n4 = new Node (4, n1.next);
n2.next.elt = 17;
return n1.elt
STACK
n1
n2
n3
HEAP
NODE
elt
1
next
null
NODE
elt
2
next
Node n1 = new Node (1, null);
Node n2 = new Node (2, n1);
Node n3 = n2;

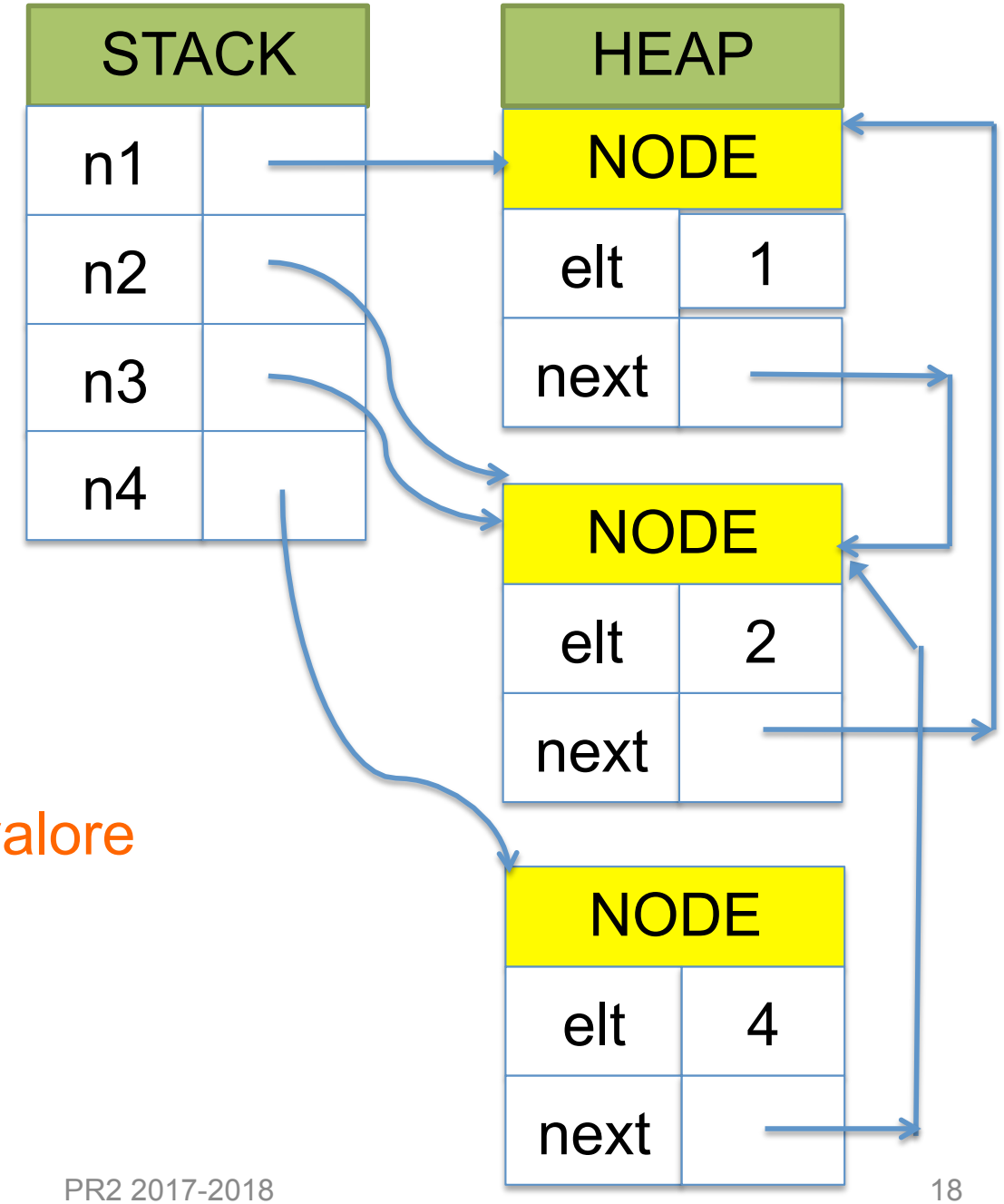
WORKSPACE

```
n3.next.next = n2;
Node n4 = new Node (4, n1.next);
n2.next.elt = 17;
return n1.elt
```

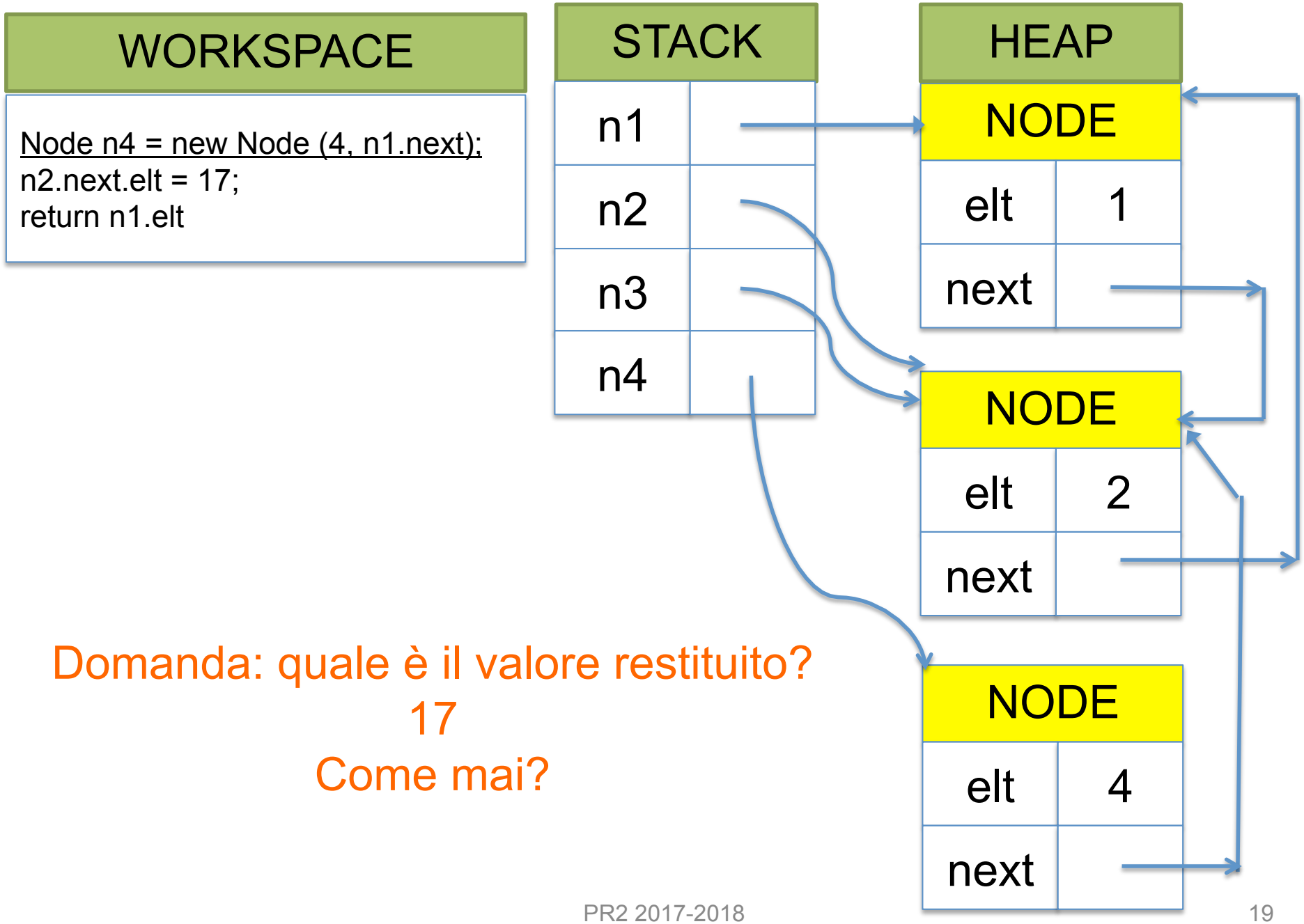


WORKSPACE

Node n4 = new Node (4, n1.next);
n2.next.elt = 17;
return n1.elt



Domanda: quale è il valore restituito?



Domanda: quale è il valore restituito?

17

Come mai?

Ereditarietà

Ereditarietà tra classi

- Esamineremo solo la nozione di ereditarietà tra classi
- La nozione di ereditarietà vale anche per le interfacce: dettagli nelle note on-line (e in ogni manuale)
- Strumento tipico dell'OOP per riusare il codice e creare una gerarchia di astrazioni
- Generalizzazione: una super-classe generalizza una sotto-classe fornendo un comportamento che è condiviso dalle sotto-classi
- Specializzazione: una sotto-classe specializza (concretizza) il comportamento di una super-classe

Perché è importante

- Permette di specializzare il comportamento di una classe, prevedendo nuove funzionalità, ma al tempo stesso mantendendo le vecchie e quindi senza influenzare codice cliente già scritto
- Riuso del codice!!
- Subtype polymorphism: *tramite l'ereditarietà un variabile può assumere tipi (di classi) differenti*
- Una funzione con parametro formale di tipo T può operare con un parametro attuale di tipo S a patto che S sia un sotto-tipo di T

Subtyping

- B è un sotto-tipo di A: “every object that satisfies interface B also satisfies interface A”

- Obiettivo metodologico: il codice scritto guardando la specifica di A opera correttamente anche se viene usata la specifica di B

Sotto-tipi e principio di sostituzione

- B è sotto-tipo di A: B può essere sostituito per A
  - una istanza del sotto-tipo soddisfa le proprietà del super-tipo
  - una istanza del sotto-tipo può avere maggiori vincoli di quella del super-tipo

Sotto-tipo nozione semantica

- Sotto-tipo è una nozione semantica
  - B è un sotto-tipo di A se (e solo se) un oggetto di B si può mascherare come uno di A in tutti i possibili contesti
- Ereditarietà è una nozione di implementazione
  - creare una nuova classe evidenziando solo le differenze (il codice nuovo)

Ereditarietà in Java

- Una **sotto-classe** si definisce usando la parola chiave extends
  - **class B extends A { … }**
- Osservazione: l'ereditarietà in Java è semplice!
  - una classe può implementare più interfacce, ma estendere solo una super-classe
  - questo non vale in altri linguaggi a oggetti: l'esempio classico è C++

Esempio

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

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

La classe C eredita implicitamente i campi definiti dalla classe D

Nella classe D non sono visibili le variabili e i 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, y;
      public int addBoth( ) { return x + y; }
}

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

Nella classe D sono visibili le variabili e i metodi dichiarati **protected**. Quindi sono riferibili direttamente... ma in questo caso la soluzione precedente era metodologicamente migliore

Metodo costruttore e Super

- Un aspetto critico della nozione di ereditarietà è che il metodo costruttore non viene ereditato
- Tipicamente il metodo costruttore della sotto-classe deve accedere anche alle variabili di istanza della super-classe
- 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 è un sotto-tipo 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** è un riferimento all'oggetto corrente: l'oggetto il cui metodo o costruttore viene chiamato

this (esempio per disambiguare)

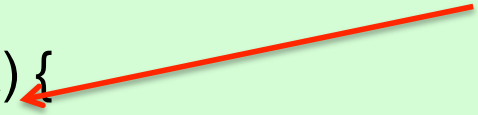
```
public class Point {
      private int x = 0;
      private int y = 0;

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

```
public class Point {
      private int x = 0;
      private 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 {
    private 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;
    }
}
```

Non può esser sovrascritto!

```
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"));
    }
```

Si inizializza solo con i costruttori!

Cosa stampa?

Esempio

```
public class Point {
    private 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 stampa? [4,2]:null

Upcasting & downcasting

- Supponiamo che T sia una sotto-classe di S (in una gerarchia)
- **Upcasting**: un oggetto di tipo T può essere legato a una variabile di tipo S
- **Downcasting**: un oggetto di tipo S può essere legato a una variabile di tipo T

```
class Vehicle { … };
class Car extends Vehicle; // Car sotto-tipo di Vehicle

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

Upcasting&downcasting

- Upcasting è 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 (stesso riferimento)
  - 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 è presente 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

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

`toString`

- In Object genera una stringa contenente il tipo dell'oggetto e il suo hash code
- Normalmente si vorrebbe ottenere una stringa composta da
  - tipo
  - valori dello stato
- Se se ne ha bisogno, va sempre ridefinita