



PROGRAMMAZIONE 2

4. Un modello operativo per Java

PR2 2017-2018

1



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

PR2 2017-2018

2

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;
}
:
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 esemplificate dalla "annotazione di tipo" **Node** memorizzate nello heap. Il perché si capirà in seguito!!!

PR2 2017-2018

5

Allocazione di array sullo heap



```
int[] anArray;
anArray = new int[4];
anArray[0] = 1;
:
anArray[3] = 4;
```

HEAP			
int[]			
length			4
1	2	3	4

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

Nota importante: a run-time sono presenti informazioni di tipo esemplificate dal "tipo" array di interi e la dimensione (length) memorizzate nello heap

PR2 2017-2018

6

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;
}
```

PR2 2017-2018

7

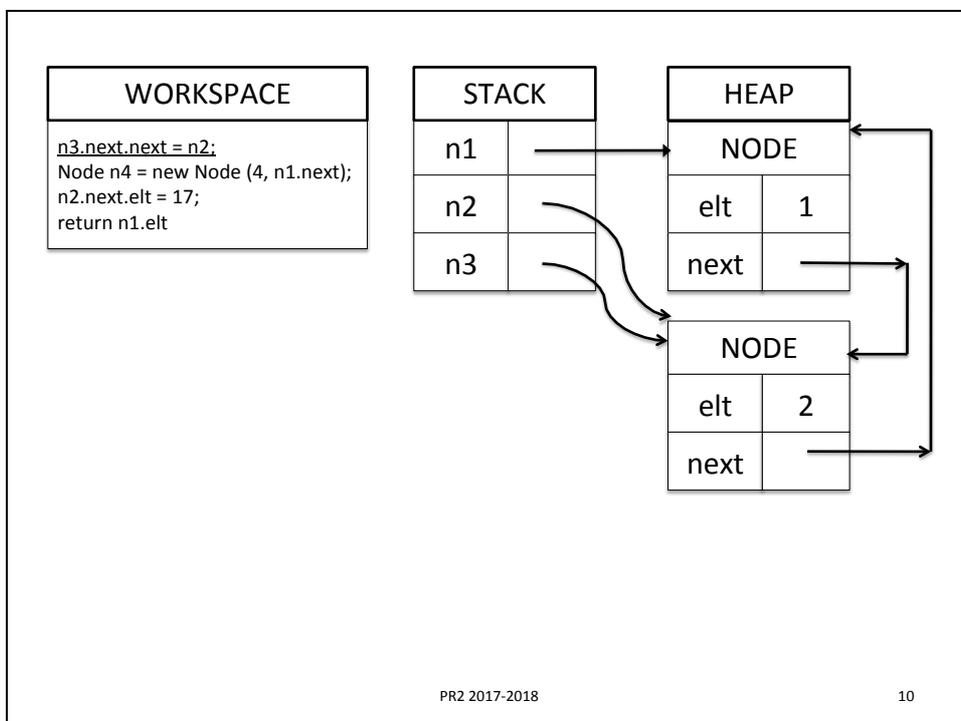
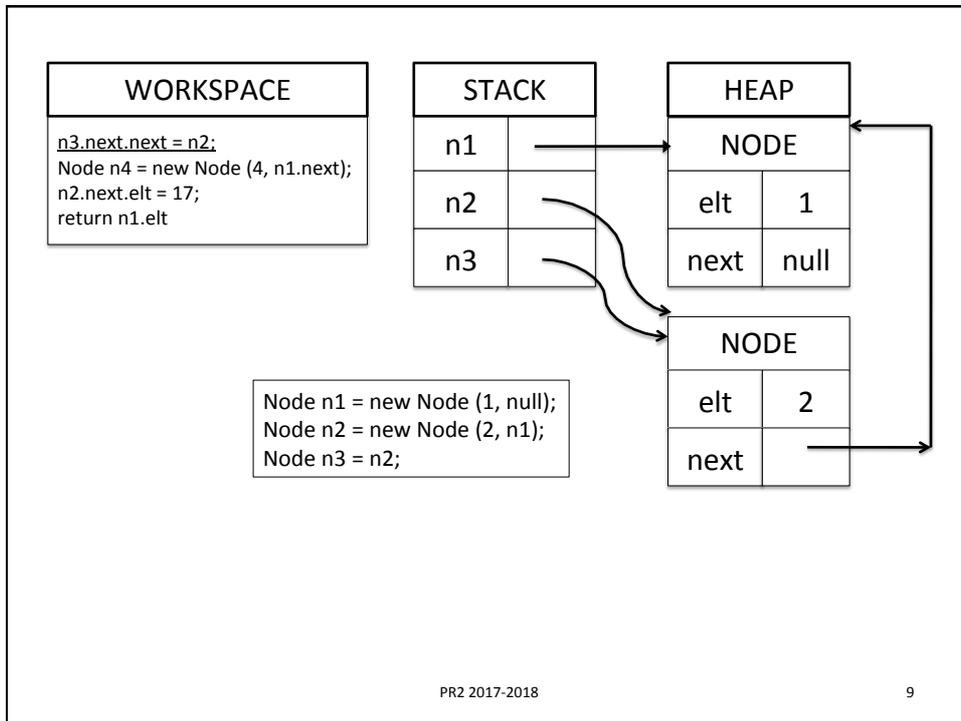
Esempio di ASM

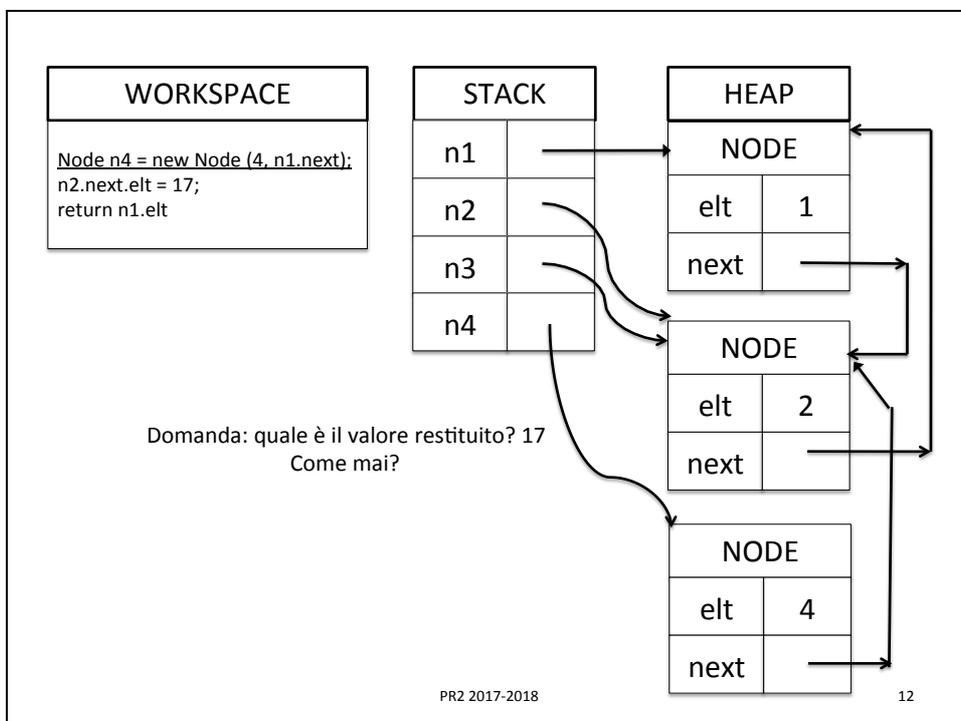
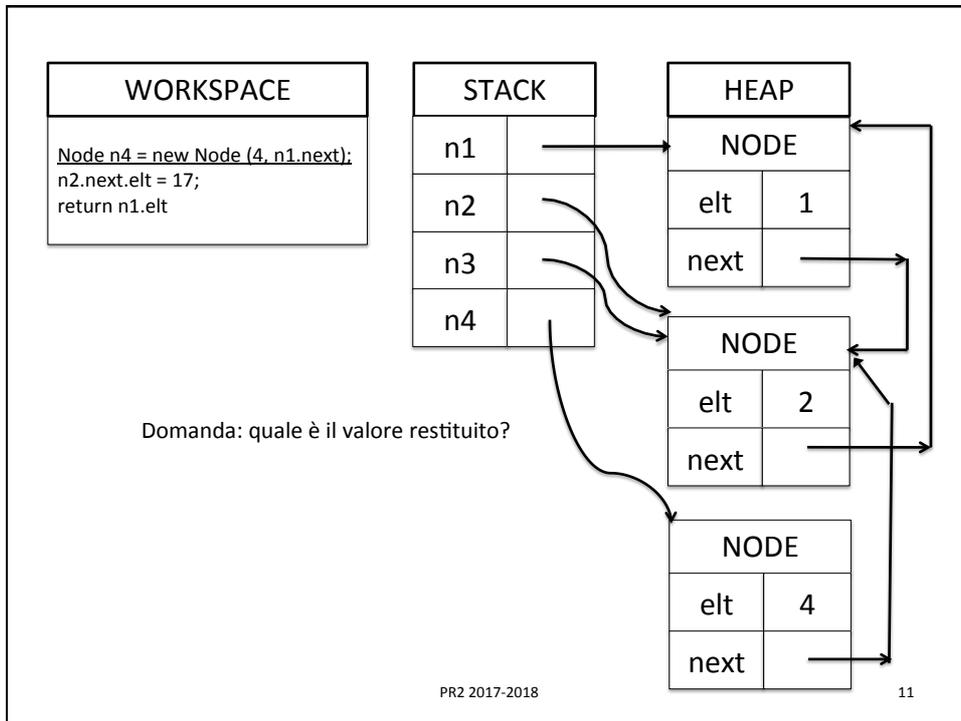


- Supponiamo di voler valutare l'invocazione
 - Node.m();
- La prima cosa da osservare è che l'invocazione del metodo restituisce un valore intero (che non è un oggetto)
- Lo stack deve contenere lo spazio per memorizzare il valore restituito dall'invocazione del metodo
 - intuitivamente una variabile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

PR2 2017-2018

8







Ereditarietà

PR2 2017-2018 13



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

PR2 2017-2018 14

Perché è importante



- Permette di specializzare il comportamento di una classe, prevedendo nuove funzionalità, ma al tempo stesso mantenendo le vecchie e quindi senza influenzare codice cliente già scritto
- Riutilizzo 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
- Questo non è sempre vero in Java

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)
- Ora esamineremo gli aspetti concreti di ereditarietà in Java: in seguito vedremo la parte semantica

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; }
}
```

La classe C eredita implicitamente i campi definiti dalla classe D

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

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; }
}
```

La classe C eredita implicitamente i campi definiti dalla classe D

```
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;
    }
}
```

PR2 2017-2018

25

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

PR2 2017-2018

26

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"));
    }
}
```

Non può esser sovrascritto!

Si inizializza solo con i costruttori!

Cosa stampa?

PR2 2017-2018 27

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"));
    }
}
```

Non può esser sovrascritto!

Si inizializza solo con i costruttori!

Cosa stampa? [4,2]:null

PR2 2017-2018 28

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



Tipo statici e dinamici

PR2 2017-2018 35



Tipi

- Il tipo **statico** di una variabile è 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 è il tipo statico della variabile c

PR2 2017-2018 36

Tipi statici e dinamici



- Il tipo **statico** di una espressione è 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 è il tipo della classe di cui l'oggetto è istanza

Statico vs. Dinamico



- Nel caso di OCaML la differenza tra tipi statici e tipi dinamici non aveva una utilità esplicita
- La presenza della nozione di ereditarietà fa emergere chiaramente questa nozione
- Il tipo dinamico di una variabile o di una espressione è sempre un sotto-tipo 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 ( );
Shape s1 = p; // Linea A
Shape s2 = c; // Linea B
s2 = p; // Linea C
```

PR2 2017-2018

39

Quesito



```
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 è il tipo statico di s1 alla linea A?

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

PR2 2017-2018

40

Quesito



```
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 è il tipo statico di s1 alla linea A?
Shape

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

PR2 2017-2018

41

Quesito



```
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

PR2 2017-2018

42



Quesito

```
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 è il tipo statico di asShape(p)?

Shape

Quale è il tipo dinamico di asShape(p)?

Point