

# **Programmazione Orientata agli Oggetti**

---

Gestione della Memoria

# Sommario

- Allocazione di Oggetti
  - L'operatore **new**
  - Costruttori
  - Costruttore di default
- Stack e Heap
  - Stack e Record di Attivazione
  - Stack-Overflow
  - Heap
- Equivalenza di oggetti ed identità dei riferimenti
- Riferimenti in Java vs Puntatori in C
- Gestione della Memoria
  - *Garbage Collection*

# Operatore new (1)

- La creazione di oggetti in Java è permessa dall'operatore **new**
  - richiede la specifica della classe di cui si vuole creare una nuova *istanza*
  - ovvero, più precisamente, di uno dei *costruttori* di tale classe
  - restituisce un *riferimento* all'oggetto appena creato

- `Punto origine = new Punto();`

```
public class Punto {  
    private int x;  
    private int y;  
    public Punto() {  
        this.x = 0;  
        this.y = 0;  
    }  
    ...  
}
```

*Definizione del Costruttore  
(sintassi: stesso nome della classe)*

*Corpo del costruttore {...}*

*File* `Punto.java`

- Attenzione: `origine` contiene un *riferimento*; NON contiene l'oggetto appena creato

# Operatore new (2)

- Solo la JVM può creare oggetti
  - l'operatore **new** consente di chiederne i servizi
- Per inizializzare nuovi oggetti, sinora:
  - invocazione del cosiddetto *costruttore senza argomenti*  
**new Punto()** ;
  - invocazione dei vari metodi *setter*
- Ad esempio:  

```
Punto zeroUno = new Punto();  
zeroUno.setX(0);  
zeroUno.setY(1);
```
- Si possono definire *costruttori* che ricevono parametri

# Costruttori

- Ogni classe ha *sempre* (almeno) un costruttore che viene eseguito ogni volta che un oggetto di tale classe viene creato
  - ✓ altrimenti non sarebbe possibile creare oggetti
- Il corpo del costruttore costruisce lo stato iniziale di un nuovo oggetto
  - riceve informazioni sullo stato iniziale da costruire tramite i parametri
  - fissa i valori iniziali delle variabili di istanza

# Sintassi dei Costruttori

- I costruttori si denotano con lo *stesso* nome della classe in cui compaiono
- A differenza dei metodi, *NON* possono né devono specificare il tipo del valore restituito
  - N.B. *NON* è nemmeno possibile usare `void`

```
public class Punto {  
    private int x;  
    private int y;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

- ✓ N.B. I costruttori *NON* sono niente affatto metodi
  - Le differenze non sono limitate al solo tipo restituito

# Invocazione dei Costruttori

- Attraverso l'operatore **new** è possibile invocare il costruttore e specificare degli argomenti
- Ad esempio:

*Parametri attuali del costruttore*



```
Punto origine = new Punto(0, 0);
```

```
System.out.println(origine.getX()); // Stampa 0
```

# Oggetti in Corso di Costruzione

- Il corpo del costruttore è adibito alla costruzione dello stato iniziale dell'oggetto creato
- Dal momento dell'invocazione dell'operatore per la creazione di un nuovo oggetto
  - `new Punto()`  
al momento in cui l'esecuzione del corpo del costruttore è completamente terminata, l'inizializzazione dello stato non è completa
- Durante la costruzione l'oggetto transita per una serie di stati intermedi, ovvero è *inconsistente*
  - è *buona regola* non farsi mai “sfuggire” riferimenti ad oggetti in uno stato potenzialmente inconsistente (>>)



# Esempio (*con Eclipse*)

- Implementare il costruttore della classe **Rettangolo**
    - I parametri servono a specificare
      - vertice
      - base
      - altezza
- dell'oggetto appena creato

# Esempio (2)

```
public class Rettangolo {  
  
    private Punto vertice;  
    private int base;  
    private int altezza;  
  
    public Rettangolo(Punto v, int base, int altezza) {  
        this.vertice = v;  
        this.base = base;  
        this.altezza = altezza;  
    }  
    // soliti getter  
}
```

# Costruttore di Default (1)

- Tutte le classi devono avere almeno un costruttore, *sempre*
- Se non viene esplicitamente dichiarato, ne viene aggiunto uno *implicitamente*
- E' un costruttore senza parametri (*costruttore no-args*)
- Anche senza dichiarare esplicitamente il costruttore senza argomenti di **Punto**:  
**Punto punto = new Punto() ; //COMPILA**

# Costruttore di Default (2)

- Appena viene definito esplicitamente un costruttore, il *costruttore no-args* non viene più generato automaticamente e non è più possibile invocarlo
- Ad esempio:
  - Dopo aver dichiarato un primo costruttore (con parametri) nella classe **Rettangolo**:

```
public class MainNoArgs {  
    public static void main(String[] args) {  
        Rettangolo rect = new Rettangolo(); //NON COMPILA  
    }  
}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:

**The constructor Rettangolo() is undefined**  
at MainNoArgs.main(MainNoArgs.java:4)

# Variabili di Istanza: Valore di default e `null`

- È fortemente consigliato l'uso di un costruttore che inizializzi le variabili di istanza di tipo riferimento
  - Onde evitare `NullPointerException`
  - Se non specificato altrimenti, le variabili di istanza di tipo riferimento sono inizializzate a `null`
- Altro possibile costruttore per la classe `Rettangolo`:

...

```
public Rettangolo(int base, int altezza) {  
    this.vertice = new Punto(0, 0);  
    this.base = base;  
    this.altezza = altezza;  
}
```

...

# Costruttori Alternativi

- Se il vertice non viene specificato, questo costruttore suppone che sia nell'origine
  - è poi possibile usare la variabile di istanza **vertice** senza sollevare **NullPointerException**
- Spesso si definiscono molteplici costruttori con diversi parametri
  - Utili per costruire oggetti
    - a partire da informazioni diverse. Ad es. per i rettangoli
      - base, altezza, vertice alto a sx
      - vertice alto a sx, vertice basso a dx
    - senza essere costretti a specificare tutti i parametri
  - Per ora: unico costruttore, il più generico possibile (>>)

# Gestione della Memoria: Stack e Heap

- Durante l'esecuzione, un programma ha accesso ad almeno due distinte aree di memoria
- *Stack*

Per la memorizzazione delle informazioni necessarie per l'esecuzione dei metodi, anche nidificati

  - Conserva:
    - Stato dell'esecuzione
    - Variabili locali e loro valore
- *Heap*

Per la memorizzazione di oggetti creati tramite l'operatore **new**

  - Conserva:
    - Oggetti e loro stato

# Stack

- Area di memoria assegnata dalla JVM all'inizio dell'esecuzione
  - dimensione massima prefissata (qualche *mb*)
- Adibita a conservare quanto serve per mantenere lo stato dell'esecuzione (tranne lo stato degli oggetti >>)
  - È una struttura dati gestita secondo una disciplina LIFO
    - *Last In First Out*
    - Per gestire le invocazioni di metodo, comunque nidificate, a cominciare dal `main()`
- Contiene i *Record di Attivazione (RDA)*
  - Struttura dati che contiene tutte le informazioni necessarie all'invocazione ed all'esecuzione dei metodi



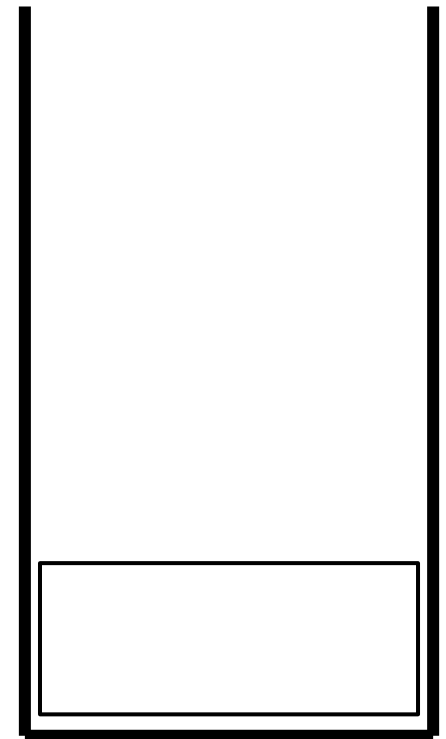
# Record di Attivazione

- Ogni volta che un metodo viene invocato, il corrispondente RDA viene creato ed inserito in cima allo stack
- Quando il metodo termina, il suo RDA viene rimosso
- La gestione dello stack avviene in modo automatico e trasparente da parte della JVM

# Metodi e Record di Attivazione (1)

```
public static void main(String[] args) { ←  
    ...  
    oggetto1.metodo1();  
}  
  
// ... (nella classe di "oggetto1")  
  
public void metodo1() {  
    oggetto2.metodo2();  
}  
...
```

*RDA main*



Stack

# Metodi e Record di Attivazione (2)

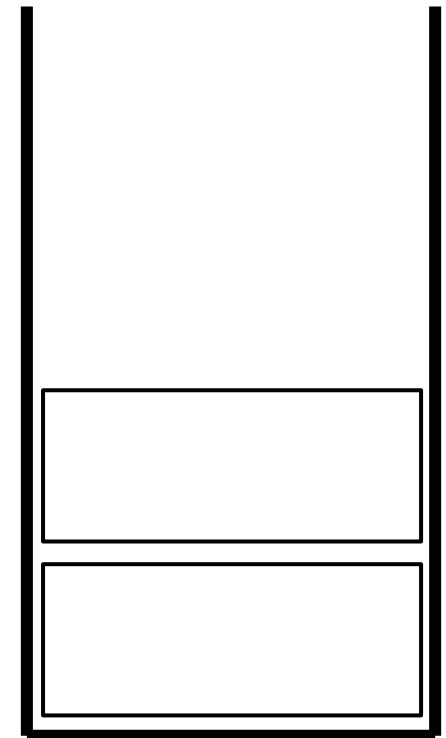
```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1(); ←  
}
```

// ... (nella classe di "oggetto1")

```
public void metodo1() {  
    oggetto2.metodo2();  
}  
...
```

*RDA metodo1*

*RDA main*



Stack

# Metodi e Record di Attivazione (3)

```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```

// ... (nella classe di "oggetto1")

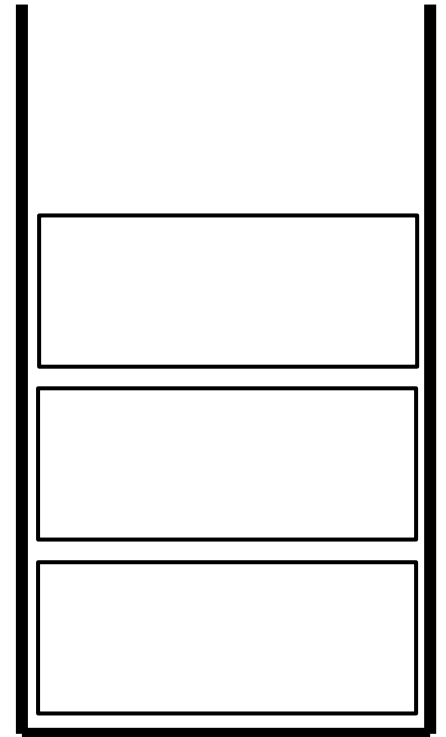
```
public void metodo1() {  
    oggetto2.metodo2();  
}
```

...

*RDA metodo2*

*RDA metodo1*

*RDA main*



Stack

# Metodi e Record di Attivazione (4)

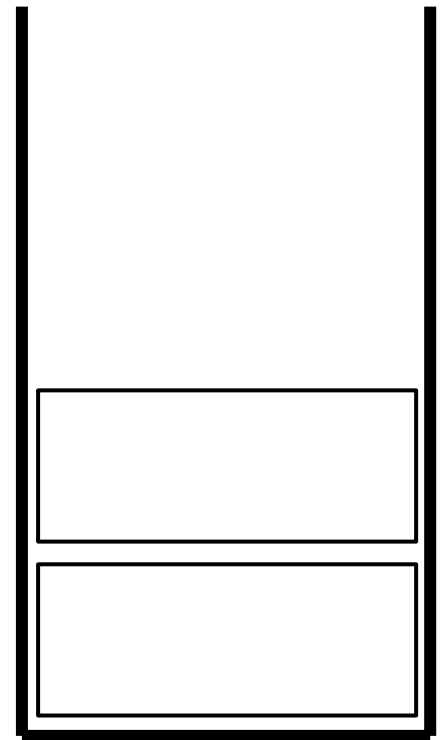
```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```

// ... (nella classe di "oggetto1")

```
public void metodo1() {  
    oggetto2.metodo2();  
    ←  
}  
...
```

*RDA metodo1*

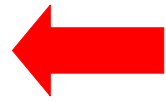
*RDA main*



Stack

# Metodi e Record di Attivazione (5)

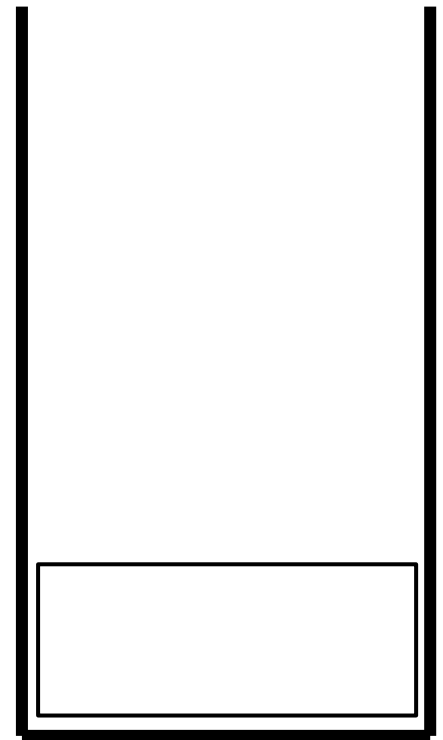
```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```



// ... (nella classe di "oggetto1")

```
public void metodo1() {  
    oggetto2.metodo2();  
}  
...
```

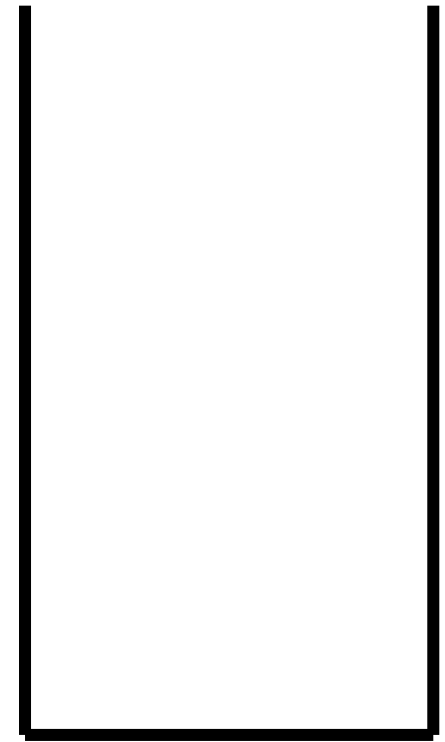
*RDA main*



Stack

# Metodi e Record di Attivazione (6)

```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}  
// ... (nella classe di "oggetto1")  
  
public void metodo1() {  
    oggetto2.metodo2();  
}  
...
```



Stack

# Contenuto di un *RDA*

- Un *Record di Attivazione* contiene:
  - parametri attuali
  - eventuale riferimento all'oggetto corrente per invocazione (**this**)
  - variabili locali
  - valore di ritorno del metodo (se non è **void**)
  - il punto di ritorno dell'invocazione di metodo: l'istruzione successiva all'invocazione nel metodo chiamante



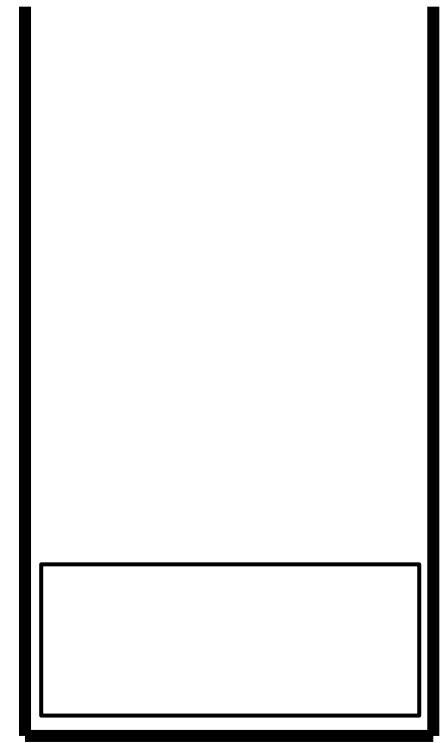
# Variabili Locali dentro il RDA

- Il record di attivazione ospita anche le variabili locali di ogni metodo
- Lo *scope* delle variabili locali è il metodo in cui sono definite, ed il ciclo di vita è chiaramente quello dell'invocazione di metodo
- L'*identificatore* di una variabile locale è un *alias* (gestito dal compilatore e dall'ambiente di esecuzione) di un indirizzo di memoria relativo (sullo stack) in cui è conservato il suo valore
  - Attenzione: Nulla a che fare con i riferimenti
  - Gli identificatori (ad es. **origine**) sono decisamente più facili da ricordare rispetto ad un indirizzo di memoria!

# Stack & Variabili Locali (1)

```
public static void main(String[] args) { ←  
    ...  
    oggetto1.metodo1();  
}  
  
// ... (nella classe di "oggetto1")  
  
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
}  
...
```

*RDA main*



Stack

# Stack & Variabili Locali (2)

```
public static void main(String[] args) {
```

```
...
```

```
    oggetto1.metodo1();
```



```
}
```

```
// ... (nella classe di "oggetto1")
```

```
public void metodo1() {
```

```
    int contat = 3;
```

```
    oggetto2.metodo2();
```

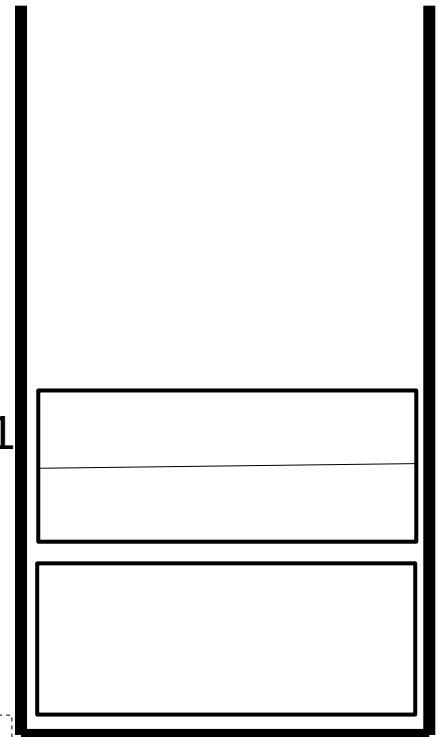
```
    contat = 5;
```

```
}
```

```
...
```

*RDA metodo1*  
contat: 0x34fa09b1

*RDA main*



Stack

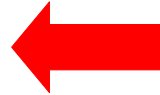
*contat* è in realtà un alias per un indirizzo di memoria relativo nel *record di attivazione*

# Stack & Variabili Locali (3)

```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```

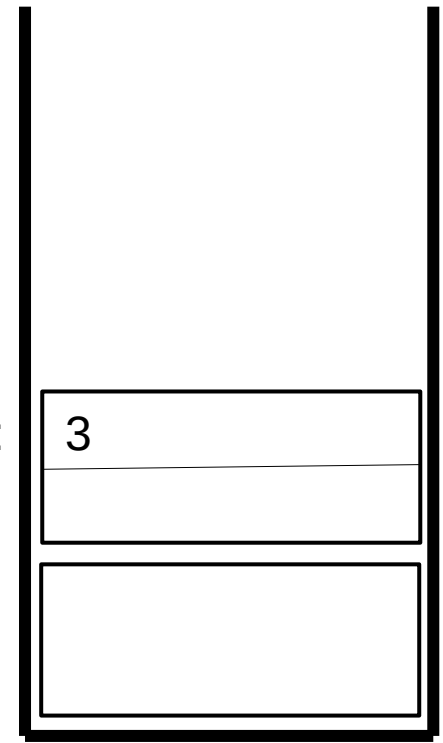
// ... (nella classe di "oggetto1")

```
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
}  
...
```



*RDA metodo1*  
contat:

*RDA main*



Stack

# Stack & Variabili Locali (4)

```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```

// ... (nella classe di "oggetto1")

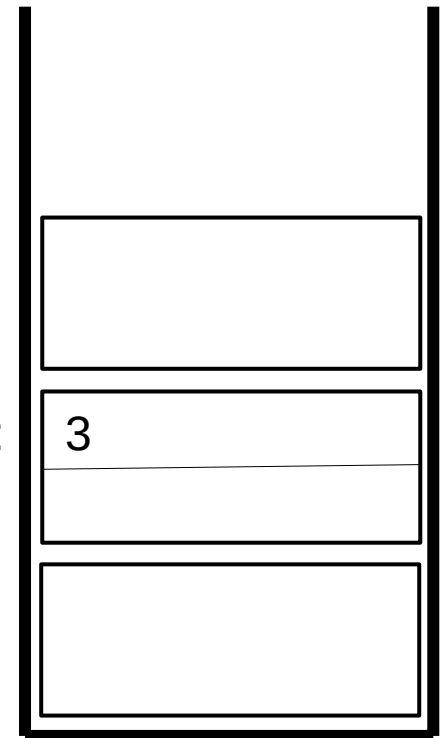
```
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
}
```

...

*RDA metodo2*

*RDA metodo1*  
contat:

*RDA main*



Stack

# Stack & Variabili Locali (5)

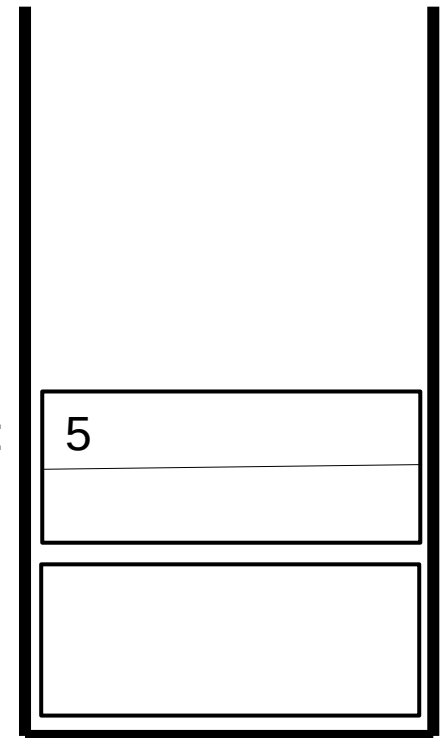
```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```

// ... (nella classe di "oggetto1")

```
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
}  
...
```



*RDA metodo1*  
contat:  
  
*RDA main*

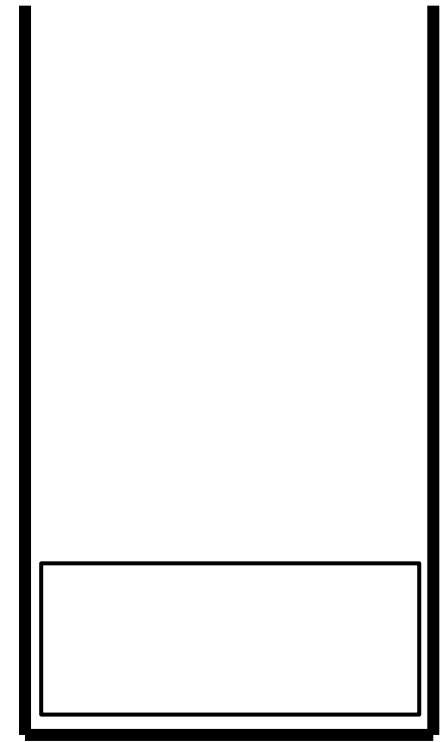


Stack

# Stack & Variabili Locali (6)

```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}  
  
// ... (nella classe di "oggetto1")  
  
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
} ←  
...
```

*RDA main*



Stack

# Stack & Variabili Locali (7)

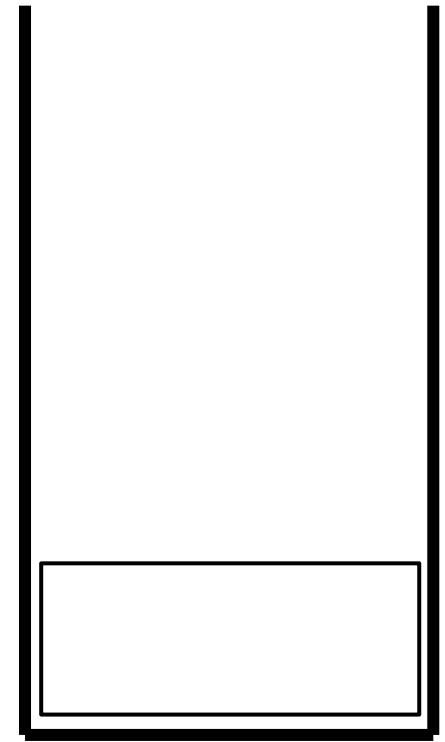
```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}
```



```
// ... (nella classe di "oggetto1")
```

```
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
}  
...
```

*RDA main*

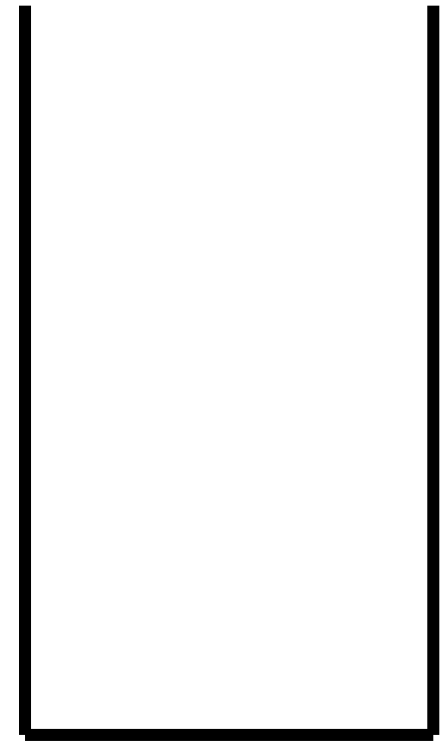


Stack



# Stack & Variabili Locali (8)

```
public static void main(String[] args) {  
    ...  
    oggetto1.metodo1();  
}  
←  
// ... (nella classe di "oggetto1")  
  
public void metodo1() {  
    int contat = 3;  
    oggetto2.metodo2();  
    contat = 5;  
}  
...
```



Stack

# Scope delle Variabili Locali

- Il primo record di attivazione è sempre quello del metodo `main()`
- Quando un *RDA* viene rimosso dallo stack le variabili locali e i parametri non sono più utilizzabili
  - Lo *scope* di queste informazioni è limitato al solo metodo di appartenenza

# Stack Overflow (1)

- Lo stack ha una dimensione limitata
  - non può contenere troppi RDA
- Ad esempio

```
public class SommatoreRicorsivo {  
    public int sommaDaZeroA(int n) {  
        return n + sommaDaZeroA(n-1);  
    }  
}
```

***// ERRORE! manca il caso base!***

# Stack Overflow (2)

- L'esecuzione del metodo `sommaDaZeroA(1)` genera il seguente errore (indipendentemente dal suo argomento)

**Exception in thread "main"**

**`java.lang.StackOverflowError`**

```
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
...
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
at SommatoreRicorsivo.sommaDaZeroA(SommatoreRicorsivo.java:3)
...
```

N.B. La diagnostica riporta nomi delle classi e dei metodi i cui RDA sono nello stack al momento del *trabocco*

# Heap

- Tutti gli oggetti creati con l'operatore **new** possiedono uno stato che viene conservato in un'area di memoria denominata *Heap*
  - Letteralmente “*mucchio*”, si tratta di un'area di memoria assegnata dalla JVM
  - NON è gestito come lo stack (LIFO)
    - la sua dimensione può crescere e decrescere dinamicamente anche durante l'esecuzione
    - gli oggetti sono creati e deallocati in ordine sparso
    - spesso e volentieri l'occupazione dell'heap risulta molto frammentata

# new VS malloc

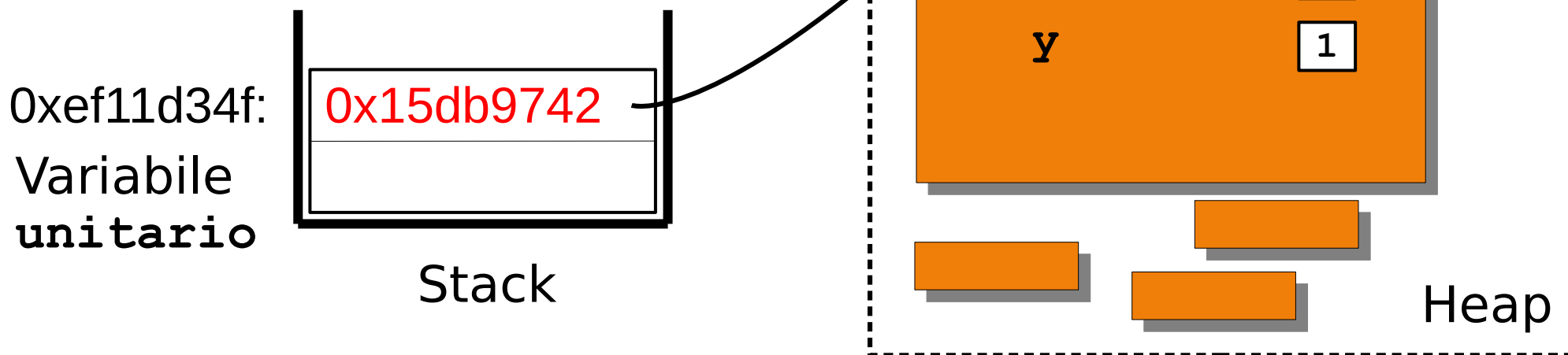
- **new** sta ad una classe (quasi) come **malloc** sta ad una **struct**
  - **malloc**
    - **alloca memoria nello heap**
    - restituisce l'indirizzo di tale area di memoria
  - **new**
    - **alloca memoria nello heap**
    - restituisce il riferimento all'oggetto appena allocato
    - invoca un costruttore

# Heap e Stack (1)

- Il valore restituito dall'operatore `new` è il riferimento all'oggetto appena creato

```
Punto unitario = new Punto(1, 1)
```

- La variabile locale `unitario` contiene il riferimento all'oggetto creato, *NON* l'oggetto stesso
- `unitario` è un alias per l'indirizzo `0xef11d34f`



# Heap e Stack (2)

- È possibile stampare anche il contenuto di una variabile che contiene un riferimento ad oggetto usando il metodo `println()`

```
System.out.println(unitario);
```

- `println()` stampa una stringa che dipende solamente dall'indirizzo di memoria e dalla classe di appartenenza

Stampa: **Punto@15db9742**

(è possibile alterare questo comportamento>>)

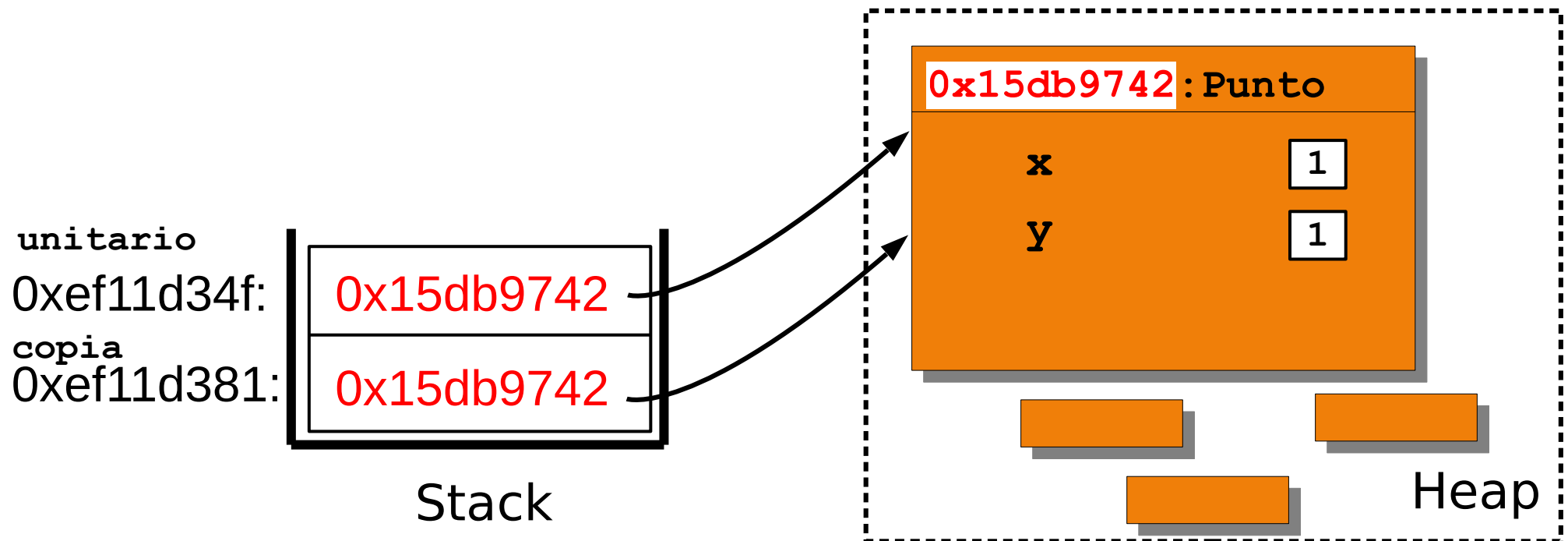


# Assegnazione di Riferimenti (1)

- L'assegnazione di una variabile che contiene un riferimento copia **il riferimento**
  - **NON** si crea un nuovo oggetto

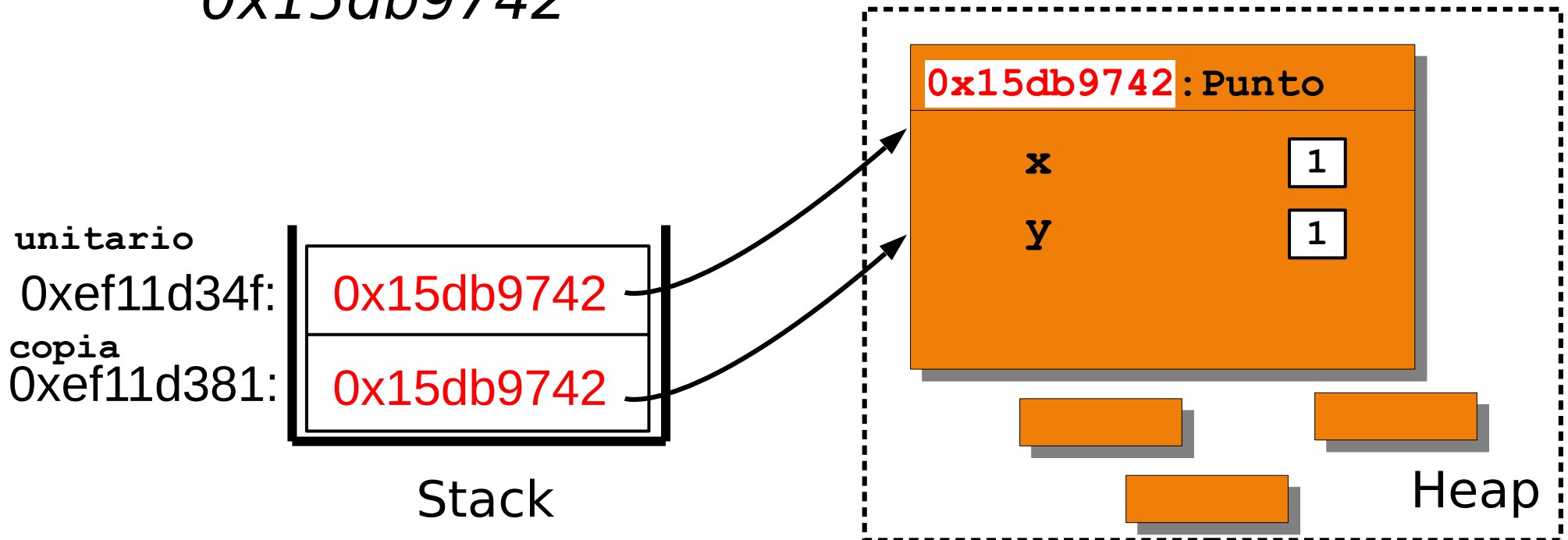
```
Punto unitario = new Punto(1, 1);
```

```
Punto copia = unitario;
```



# Assegnazione di Riferimenti (2)

- **unitario** e **copia** sono due variabili locali distinte
- Possiedono valori in posizioni distinte sullo *Stack*...
  - *0xef11d34f* e *0xef11d381*
- ...ma fanno riferimento allo stesso oggetto dentro l'*heap*, quello di indirizzo
  - *0x15db9742*



# Molteplici Riferimenti Stesso Oggetto

- Se due variabili contengono un riferimento al medesimo oggetto, ogni modifica operata ad un oggetto tramite un riferimento è anche visibile tramite l'altro riferimento
- Ad esempio:

```
Punto unitario = new Punto(1, 1);  
Punto copia = unitario;
```

```
System.out.println(copia.getX()); // Stampa 1
```

```
copia.setX(2);
```

```
System.out.println(unitario.getX()); // Stampa 2
```

# Side-Effect e Metodi

- I metodi che aggiornano lo stato di oggetti di cui ricevono un riferimento (tra i parametri) producono effetti collaterali (*side-effect*)

```
public class ModificatoreDiPunti {  
    public void azzera(Punto p) {  
        p.setX(0); p.setY(0);  
    }  
}  
  
...  
  
public static void main(String[] args) {  
    ModificatoreDiPunti m = new ModificatoreDiPunti();  
    Punto unitario = new Punto(1, 1);  
    m.azzera(unitario);  
    System.out.println(unitario.getX()); // Stampa 0  
}
```

# Equivalenza tra *Oggetti* vs Identicità dei *Riferimenti*

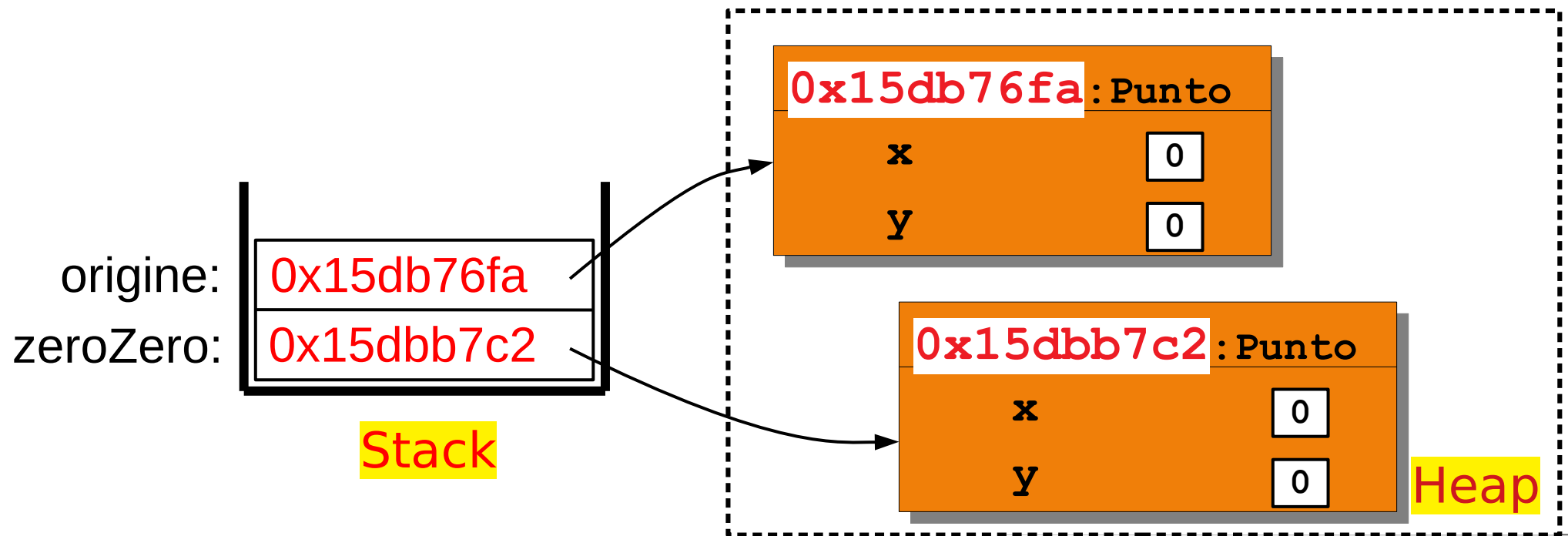
```
Punto origine = new Punto(0, 0);  
Punto zeroZero = new Punto(0, 0);
```

```
if (origine == zeroZero)  
    System.out.println("uguali");  
else  
    System.out.println("diversi");
```

- Stamperà "uguali" o "diversi"?

# Identicità dei Riferimenti (1)

- L'operatore **==** verifica se il contenuto delle due variabili è identico
  - **origine** e **zeroZero** contengono però riferimenti diversi verso due oggetti distinti (con diversi indirizzi di memoria)
  - Sono quindi considerati diversi



# Identicità dei Riferimenti (2)

- L'operatore `==`, quando applicato a variabili che contengono riferimenti ad oggetti, verifica se i riferimenti raggiungono lo stesso oggetto

```
Punto origine = new Punto(0, 0);
```

```
Punto copiaDelRif = origine;
```

```
if (origine == copiaDelRif)
```

```
    System.out.println("uguali");
```

```
else
```

```
    System.out.println("diversi");
```

- Può essere applicato anche ad espressioni di tutti i tipi primitivi, con la semantica più naturale: identità di valori
- Invece, per verificare se due oggetti distinti sono equivalenti in base al loro stato (*non* in base ai riferimenti) è necessario implementare un *criterio di equivalenza* tra oggetti di un certo tipo (>> )

# Equivalenza fra Stringhe

- Molte classi offrono un metodo `equals()` che definisce un criterio di equivalenza basato sullo stato
  - verifica l'uguaglianza con un altro oggetto passato come parametro
  - segnatura niente affatto scontata ( $>>$ )
- Per il momento conviene vedere esempi di criteri di equivalenza già definiti nelle librerie standard, come per la classe `String`
- In Java le stringhe sono oggetti a tutti gli effetti
  - è possibile confrontarle con il loro metodo `equals()`
  - equivalenza *carattere-per-carattere*

```
String nome1 = new String("alice");  
String nome2 = new String("alice");
```

```
System.out.println(nome1 == nome2);           // Stampa: false  
System.out.println(nome1.equals(nome2));      // Stampa: true
```



# Metodo `equals()`

- Il metodo `equals()` si usa quando si vuole verificare l'*equivalenza* tra due oggetti distinti secondo un criterio definito dal programmatore (e dipendente dalla classe)
  - In effetti vedremo che il metodo `equals()` è offerto, sempre, da tutte le classi (>>)
  - Ma se non viene esplicitamente implementato (>>) ha la stessa semantica dell'operatore `==` per il confronto tra riferimenti
- Se lo scopo è quello di controllare se due variabili fanno riferimento allo stesso oggetto meglio usare, sempre e comunque, il più esplicito operatore `==`

# Uguaglianza con null

- L'uguaglianza di un riferimento ad oggetti con il valore `null` si verifica tramite l'operatore `==`
  - `null` è un valore speciale che può essere utilizzato per rappresentare l'assenza di un oggetto (di un *qualsiasi tipo*)
- Tipica istruzione condizionale:

```
if (varRif!=null) {  
    <operazioni su varRif non null>  
}
```
- Attenzione: valutando `varRif.equals(null)` si ottiene
  - `false` se `varRif!=null`
  - `NullPointerException` se `varRif==null`

# Null Simmetria dell'Equivalenza

- ✓ Quindi: `a.equals(b)` *non* è, in generale, pari a `b.equals(a)`
- Comportamento atteso se `a` e `b` possono anche assumere valori `null`
- Se invece sia `a` sia `b` sono
  - non nulle
  - riferimenti ad oggetti dello stesso tipo
- È necessario che valgano le proprietà tipiche delle relazioni di equivalenza, come:
  - `a.equals(a)` *(riflessiva)*
  - `( a.equals(b) ) == ( b.equals(a) )` *(simmetrica)*
  - (altre seguiranno >>)

# *Riferimenti in Java vs Puntatori C*

- Nel linguaggio C il concetto più simile a quello di *riferimento* Java è sicuramente quello di *puntatore* ad una cella di memoria
- N.B. sono e rimangono comunque concetti molto diversi, i cui costrutti, nei due linguaggi, supportano un insieme di operazioni ben diverse
  - In C tale insieme di operazioni è molto più ampio
  - In Java non esiste l'aritmetica dei puntatori come in C o C++
  - In Java non è possibile referenziare *nulla* che non sia un oggetto

# Diagnostica: *Riferimenti vs Puntatori*

- Per colpa di un errore di programmazione, spesso si finisce per produrre puntatori “impazziti” in C
  - Questi a loro voltano, causano errori a tempo di esec.
  - In buona parte i puntatori sono alla base della pessima diagnostica a tempo di esecuzione: non si può prevedere a cosa si finisce per accedere
    - ✓ risulta difficile risalire alla vera causa di un problema se gran parte degli errori viene diagnosticato con un generico *Segmentation fault* !

# *Riferimenti vs Puntatori*

- In Java i riferimenti sono stati progettati proprio per superare i limiti ed i difetti tipici dei puntatori in C
- In Java è stato deciso di evitare alla radice i rischi tipici dei puntatori
  - Non è possibile referenziare nulla che non sia un oggetto
- Si “nasconde” al programmatore l'esistenza stessa del concetto di puntatore sostituendolo con il ben più “sicuro” riferimento ad oggetto

# Gestione della Memoria

- In Java l'operatore **new** (che serve ad allocare oggetti) è il costrutto più simile alla funzione (C) **malloc**
  - entrambi allocano un'area di memoria nell'heap
- Ad ogni chiamata della funzione **malloc** deve però corrispondere una chiamata ad una funzione **free** per deallocare l'area di memoria occupata nell'heap e restituirla ai successivi utilizzi
- In Java NON è necessario
  - finora non è stata scritta neanche una operazione per liberare la memoria occupata con le **new**
- La deallocazione della memoria a carico della JVM
  - specificatamente di un componente, il *Garbage Collector*

# Deallocazione della Memoria in C

- Deallocazione della memoria a carico del programmatore
- Possibili errori che ne conseguono, ad es.:
  - **malloc** senza corrispondente **free**  
*Memory leak*: perdita di memoria utilizzabile
  - **free** senza porre il puntatore a **NULL**  
*Dangling Pointer*: rimane disponibile un puntatore verso un'area di memoria non più disponibile
    - N.B. è possibile avere la stessa problematica ancor più semplicemente (senza **malloc**) anche creando puntatori ad aree di memoria sullo stack non più in uso!

```
struct Punto *origine;  
origine = malloc(sizeof(struct Punto));  
free(origine);  
origine->x = 0; // Comportamento indefinito
```



# Deallocazione della Memoria in Java

- La piattaforma Java solleva il programmatore dalla responsabilità di deallocare esplicitamente la memoria
  - La JVM si occupa, a tempo di esecuzione, di trovare gli oggetti inutilizzati e non più utilizzabili e recupera la loro memoria
  - Il componente della JVM che si occupa di questo compito si chiama *Garbage Collector*
- Alcuni esperti ritengono che questa sia la differenza tra i linguaggi C/Java che da sola, ed in assoluto, contribuisce in maggiore misura all'incremento di produttività dei programmatori Java

# Attenzione!

- Per prevenire fraintendimenti, meglio precisare subito che la gestione della memoria, ed un suo utilizzo appropriato, rimane comunque tra le più importanti responsabilità anche di un programmatore Java
- Il fatto che non è necessario fare esplicitamente le chiamate alla **free** non significa affatto che non si stia occupando memoria!

# Il Garbage Collector (1)

- Il ruolo del garbage collector è quello di identificare gli oggetti non utilizzati e non più utilizzabili
  - Questi oggetti devono essere marcati come *reclamabili*
  - la loro memoria recuperata per fare spazio all'allocazione di nuovi oggetti
- Come può succedere che un oggetto diventa *reclamabile*? Semplice:

```
Punto origine = new Punto(0,0) ;// occupa mem.
```

```
origine = null;
```

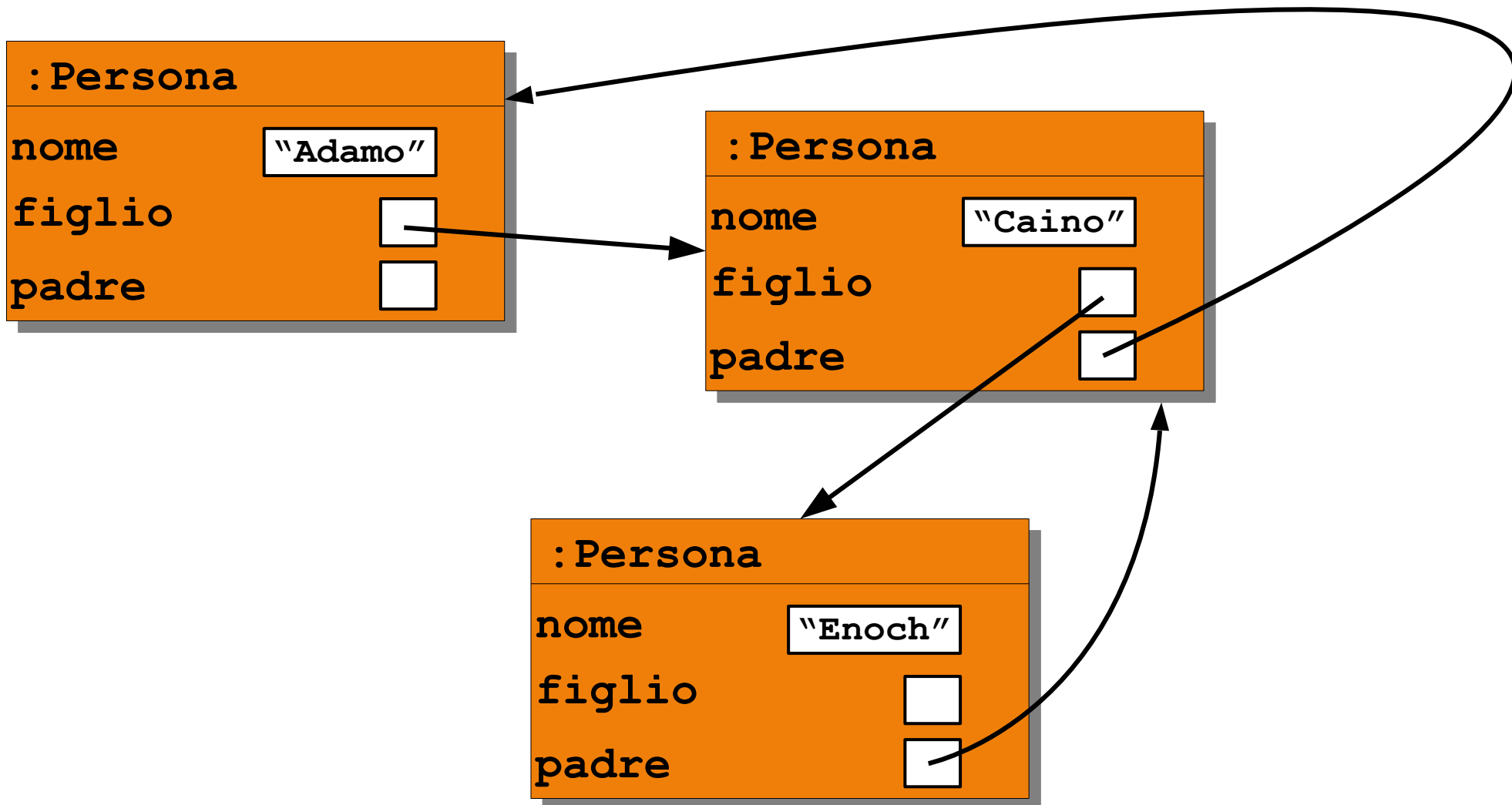
*// da qui in poi la memoria dell'oggetto può essere recuperata perché l'oggetto appena creato non è più raggiungibile*

# Il Garbage Collector (2)


- Dal punto di vista del Garbage Collector, un oggetto può essere:
  - *IN\_USO*: raggiungibile tramite una *catena di riferimenti*
  - *RECLAMABILE*: oggetto non più raggiungibile, non esiste alcuna *catena di riferimenti* che vi arrivi
- *Catena di riferimenti*: sequenza di riferimenti che parte dallo stato dell'esecuzione corrente e conduce ad un oggetto
  - eventualmente passando attraverso diversi oggetti intermedi
- Come di creano queste catene?
- Da dove iniziano?

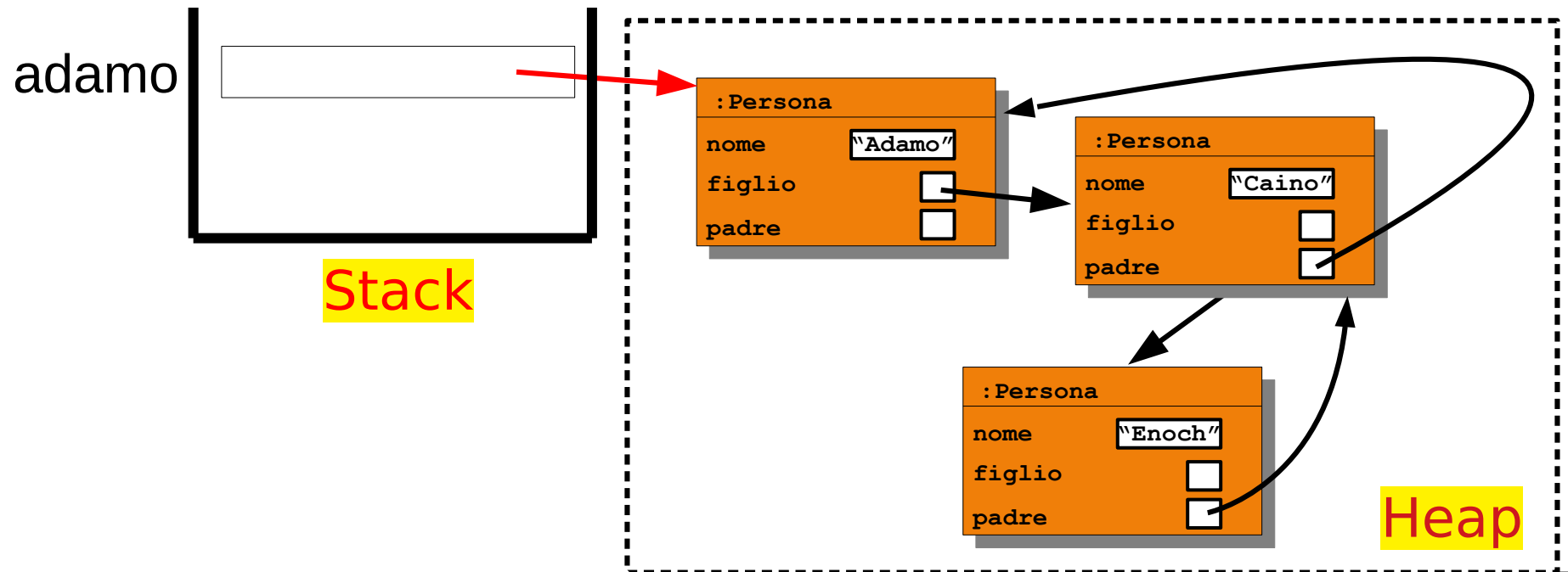
# Il Garbage Collector: Catene di Riferimenti (1)

- I riferimenti costituiscono un grafo orientato
  - Si pensi ad una possibile classe **Persona**



# Il Garbage Collector: Catene di Riferimenti (2)

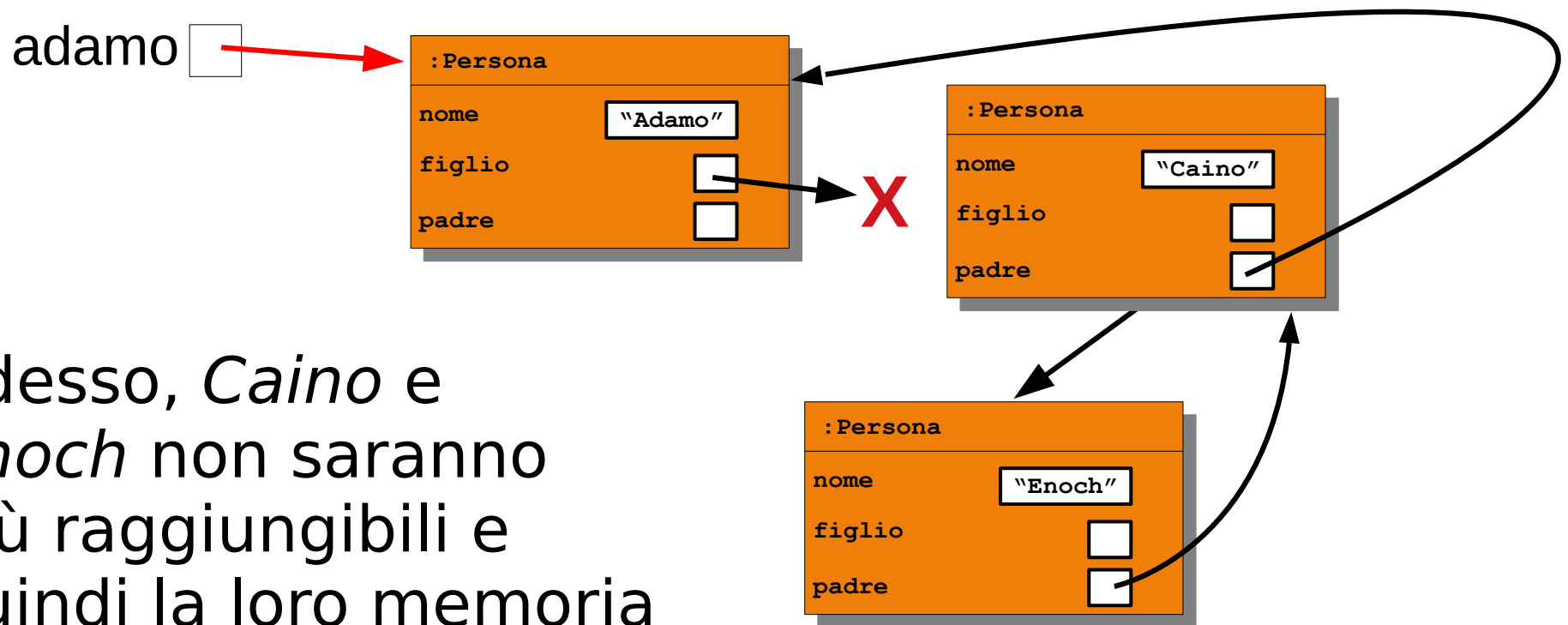
- Se la prima persona (*Adamo*) è *raggiungibile* tutte le altre mostrate nello schema sono allora raggiungibili (indirettamente tramite la prima)
- Riferimento *iniziale* (>>): 
- Se la prima persona non fosse raggiungibile, nessuna delle altre lo sarebbe



# Il Garbage Collector: Catene di Riferimenti (3)

- Per interrompere una catena di riferimenti è sufficiente porre a **null** una variabile (di istanza o locale) che contiene il riferimento ad un oggetto

`adamo.setFiglio(null);`

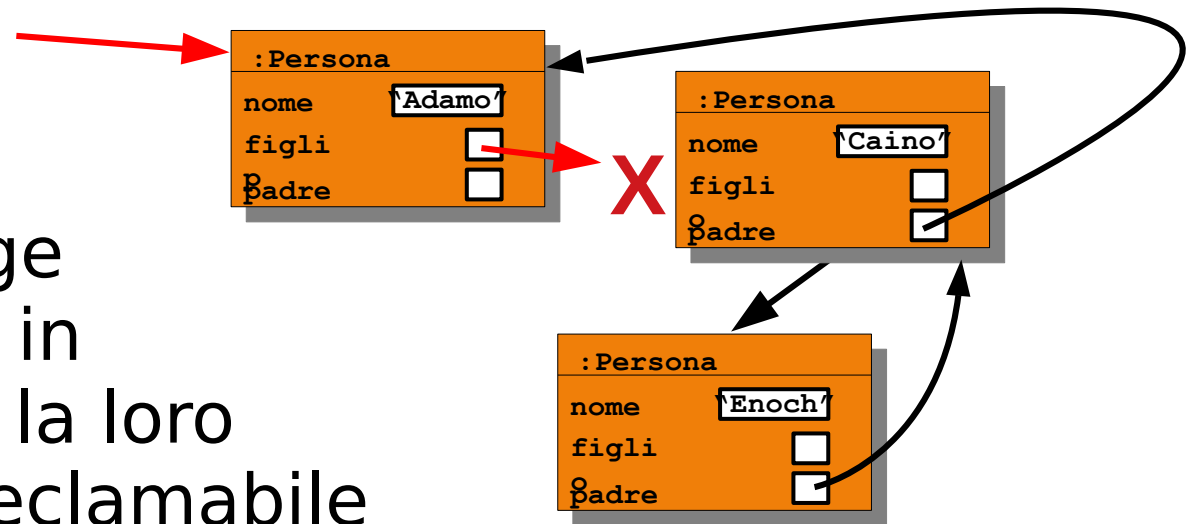


- Adesso, *Caino* e *Enoch* non saranno più raggiungibili e quindi la loro memoria può essere recuperata

# Il Garbage Collector: Catene di Riferimenti (4)

- In questo modo il Garbage Collector rileva che gli oggetti *Caino* ed *Enoch* non saranno più utilizzati
  - verranno marcati come *reclamabili*
- Mettendo a `null` il riferimento verso *Caino* non è più possibile raggiungerli

- Quando il Garbage Collector entrerà in azione marcherà la loro memoria come reclamabile



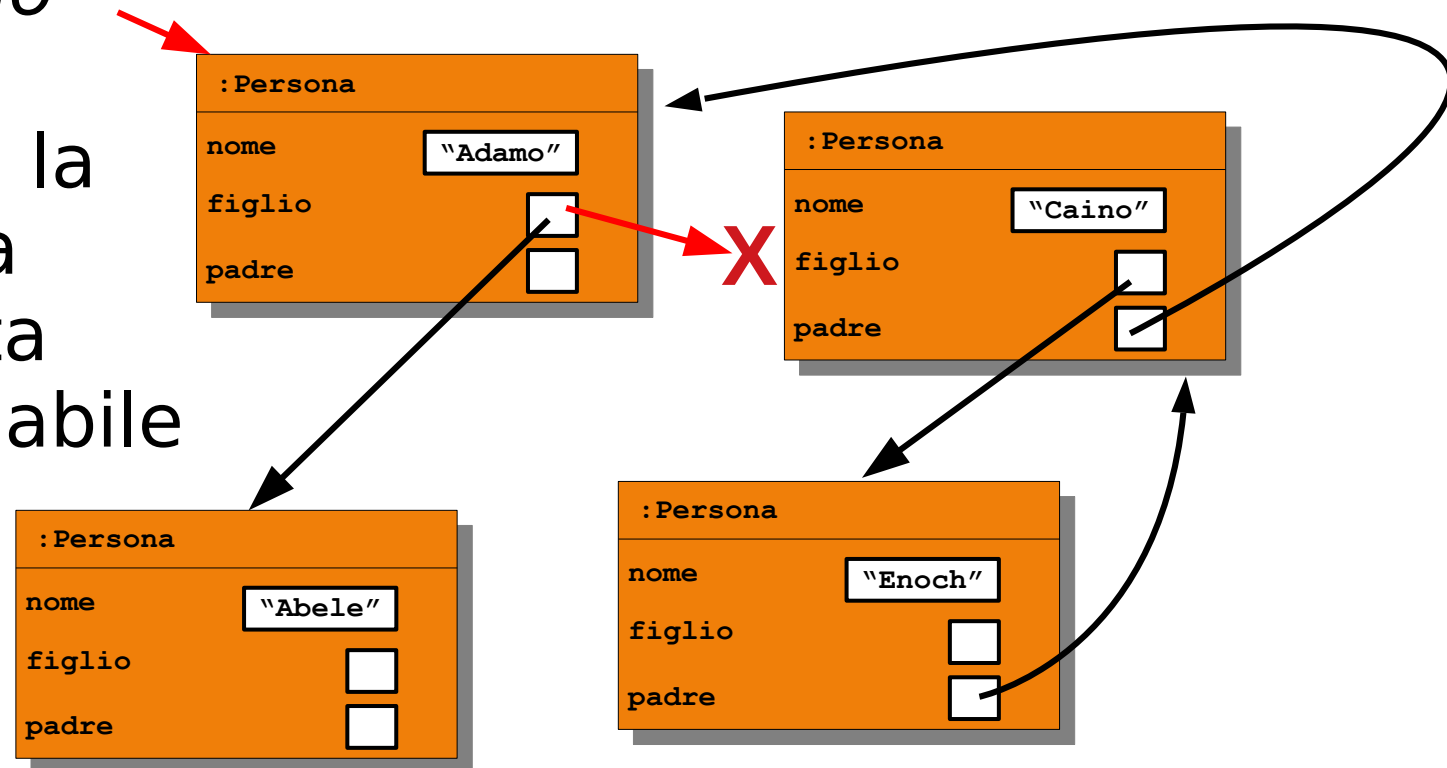


# Il Garbage Collector: Catene di Riferimenti (5)

- Un oggetto può diventare irraggiungibile anche *sovrascrivendo* un riferimento

```
Persona abele = new Persona("Abele");  
adamo.setFiglio(abele);
```

- Enoch e Caino* non sono più raggiungibili, la loro memoria verrà marcata come reclamabile



# Inizio delle Catene di Riferimenti

- Sinora si supponeva che il primo oggetto *Adamo* fosse raggiungibile
  - è necessario avere un primo riferimento per iniziare le catene
  - sia gli oggetti sia le variabili di istanza contenenti i riferimenti si trovano nell'heap
- Quali oggetti sono però raggiungibili da fuori dell'heap, e con quali riferimenti iniziali?
  - Quelli referenziati dalle variabili locali o dai parametri dei metodi i cui *RDA* sono nello Stack

# Garbage Collection (1)

- L'algoritmo lavora inseguendo i riferimenti a cominciare da quelli contenuti dentro i RDA (var. locali e parametri attuali) nello Stack
  - esplora il grafo dei riferimenti dentro l'Heap
  - continua con quelli conservati nelle var. di istanza degli oggetti conservati nell'*Heap*
  - se un oggetto è
    - *Raggiungibile*: allora viene mantenuto in memoria
    - *Non raggiungibile*: la sua memoria viene marcata *reclamabile*

# Garbage Collection (2)

- Successivamente (quando serve memoria e non necessariamente subito) la memoria reclamabile viene recuperata per poter allocare nuovi oggetti
- Abbiamo discusso solo i problemi ed i concetti basilari di una semplice implementazione dell'Algoritmo di *Garbage Collection*
  - Enorme interesse per questi algoritmi
  - Quelli utilizzati sono il risultato di decenni di ricerca (accademica ed industriale)
  - Una delle classi di algoritmi più studiate in assoluto: sempre più veloci ed ottimizzati

# Garbage Collection: Vantaggi e Svantaggi

- Facilitano lo sviluppo rapido di applicazioni
- Tuttavia esistono alcuni importanti svantaggi:
  - La visita del grafo dei riferimenti è un'operazione onerosa che limita le prestazioni
  - Il garbage collector è fuori dal controllo diretto del programmatore
    - può intervenire in momenti non facilmente prevedibili e talvolta inopportuni
    - non adatto in tutte le situazioni in cui i tempi di risposta devono essere prevedibili con certezza
- Un programmatore esperto può sicuramente ottenere prestazioni migliori gestendo la memoria manualmente ma specificatamente per la sua applicazione