UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
http://jdirocco.github.ot/

Sommario

- Incapsulamento vs information hiding
- Costruttore
- Riferimento this
- Variabili membro
- Metodi
- Package, layout sorgente e destinazione
- Modificatori di accesso
  - private
  - protected
  - public
  - package

Incapsulamento vs Information Hiding (1)

- Incapsulamento
  - **Collegare** i dati insieme con le operazioni che agiscono su di essi
- Information Hiding
  - **Nascondere** ai possibili client (utilizzatori) le scelte interne di design e gli effetti che eventuali cambiamenti di tali decisioni comportano
- Incapsulamento è una **facility** del linguaggio mentre l'information hiding è un **principio di design**

Incapsulamento vs Information Hiding (3)

VERSIONE ITALIANA

› Design Patterns - Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides,
Glossario
Incapsulamento: il risultato di nascondere una rappresentazione e un'implementazione in un oggetto. La rappresentazione non è visibile e non è possibile accedervi direttamente dall'esterno dell'oggetto. Le operazioni sono l'unico modo per accedere e modificare la rappresentazione di un oggetto

› Pagina 19
Poiché l'ereditarietà espone una sottoclasse ai dettagli dell'implementazione del genitore, si dice spesso che l'*ereditarietà rompe l'incapsulamento*

Incapsulamento vs Information Hiding (5)

VERSIONE ITALIANA

› Object-Oriented Analysis and Design with Applications by Grady Booch, Ivar Jacobson, and James Rumbaugh, pagina 51

L’incapsulamento si ottiene più spesso attraverso il meccanismo di nascondere le informazioni (non solo quello dei dati), che è il processo per nascondere tutti i segreti di un oggetto che non contribuiscono alle sue caratteristiche essenziali; in genere, la struttura di un oggetto è nascosta, così come l'implementazione dei suoi metodi

“Nessuna parte di un sistema complesso dovrebbe dipendere dai dettagli interni di qualsiasi altra parte".
Mentre l'astrazione "aiuta le persone a pensare a ciò che stanno facendo,"
l’incapsulamento "consente di apportare modifiche ai programmi in modo affidabile con uno sforzo limitato".

Incapsulamento vs Information Hiding (7)

VERSIONE ITALIANA

› Effective Java™ Third Edition by Joshua Bloch, pagina 73

Il singolo fattore più importante che distingue un modulo ben progettato da uno mal progettato è il grado in cui il modulo nasconde i suoi dati interni e altri dettagli di implementazione da altri moduli. Un modulo ben progettato nasconde tutti i dettagli dell'implementazione, separando in modo chiaro la sua API dalla sua implementazione. I moduli comunicano quindi solo attraverso le loro API e sono ignari del funzionamento interiore degli altri. Questo concetto, noto come **information hiding o incapsulamento**, è uno dei principi fondamentali della progettazione del software

Incapsulamento vs Information Hiding (9)

- **VERSIONE ITALIANA**
- Code Complete Second Edition by Steve McConnell, pagina 567 (questo libro contiene molte informazioni sull'incapsulamento, questa citazione è stata scelta perché utilizza entrambe le nozioni come sinonimi)

  L'incapsulamento (information hiding) è probabilmente lo strumento più efficace per rendere il programma gestibile e minimizzare gli effetti a catena delle modifiche al codice. Ogni volta che vedi una classe che conosce più di un'altra classe di quanto dovrebbe - comprese le classi derivate che sanno troppo dei loro genitori - errano sul lato dell'incapsulamento più forte piuttosto che più debole

Incapsulamento vs Information Hiding (10)

- Wikipedia
- Termine incapsulamento può essere usato per riferirsi a due concetti, collegati tra loro ma distinti o, a volte, alla combinazione dei due
  - un meccanismo del linguaggio di programmazione **atto a limitare l'accesso diretto** agli elementi dell'oggetto
  - un costrutto del linguaggio di programmazione che favorisce **l'integrazione (to bundle in inglese)** dei dati con i metodi (o di altre funzioni) che operano su quei dati
- Seconda definizione è motivata dal fatto che in diversi linguaggi di programmazione OO l'information hiding **non è automatico** o può essere scavalcato da altri modificatori di visibilità, pertanto l'information hiding è definito come concetto separato da chi lo preferisce alla prima

Di nuovo...

- Incapsulamento
  - **Collegare** i dati insieme con le operazioni che agiscono su di essi
- Information Hiding
  - **Nascondere** ai possibili client (utilizzatori) le scelte interne di design e gli effetti che eventuali cambiamenti di tali decisioni comportano
- Incapsulamento è una **facility** del linguaggio mentre l'information hiding è un **principio di design**

Incapsulamento vs Information Hiding (11)

- Esempio
  - Sistema che identifica un punto sulla superficie terrestre e offre funzionalità per determinare la distanza (distance) e la rotta (heading) tra due punti

Incapsulamento vs Information Hiding (12)

› Prima soluzione: No incapsulamento

```
public class Position {
      public double latitude;
      public double longitude;
}
public class PositionUtility {
   public static double distance(Position pos1, Position pos2) {
      //Calculate and return the distance between the specified positions
   }
   public static double heading(Position pos1, Position pos2) {
      //Calculate and return the heading from pos1 to pos2.
   }
}
```

Incapsulamento vs Information Hiding (13)

```
// Create a Position representing my house
Position myHouse = new Position();
myHouse.latitude = 36.538611;
myHouse.longitude = -121.797500;
// Create a Position representing a local coffee shop
Position coffeeShop = new Position();
coffeeShop.latitude = 36.539722;
coffeeShop.longitude = -121.907222;


// Use a PositionUtility to calculate distance and heading from my house to the local
coffee shop.
double distance = PositionUtility.distance(myHouse, coffeeShop);
double heading  = PositionUtility.heading(myHouse, coffeeShop);


// Print results
System.out.println ( "From my house at (" + myHouse.latitude + ", " + myHouse.longitude
+ ") to the coffee shop at (" + coffeeShop.latitude + ", " + coffeeShop.longitude + ")
is a distance of " + distance + " at a heading of " + heading + " degrees.");
```

Incapsulamento vs Information Hiding (14)

› OUTPUT

```
From my house at (36.538611, -121.7975) to the
coffee shop at (36.539722, -121.907222) is
distance of 6.0873776351893385 at a heading of
270.7547022304523 degrees.
```

› Dati (espressi mediante `Position`) ed operazioni (espresse mediante `PositionUtility`) sono separati!!!

Incapsulamento vs Information Hiding (15)

› Seconda soluzione: SI incapsulamento, NO information hiding

```
public class Position {
  public double latitude;
  public double longitude;
  public double distance(Position position) {
    // Calculate and return the distance from this object to the
       specified position.
  }
  public double heading(Position position){
    // Calculate and return the heading from this object to the
       specified position.
  }
}
```

Incapsulamento vs Information Hiding (16)

```
Position myHouse = new Position();
myHouse.latitude = 36.538611;
myHouse.longitude = -121.797500;
Position coffeeShop = new Position();
coffeeShop.latitude = 36.539722;
coffeeShop.longitude = -121.907222;
double distance = myHouse.distance(coffeeShop);
double heading = myHouse.heading(coffeeShop);

System.out.println( "From my house at (" + myHouse.latitude + ", " +
myHouse.longitude + ") to the coffee shop at (" + coffeeShop.latitude +
", " + coffeeShop.longitude + ") is a distance of " + distance + " at a
heading of " + heading + " degrees.");
```

Incapsulamento vs Information Hiding (17)

- Output identico e codice più naturale
- `myHouse.heading(coffeeShop)` calcola la rotta
  - La semantica dall'invocazione indica chiaramente quale è la direzione ovvero da `myhouse` a `coffeeShop`
- `PositionUtility.heading(myHouse, coffeeShop)` non indica chiaramente la direzione di calcolo
  - Lo sviluppatore deve ricordarsi che la funziona calcola la rotta dal primo parametro al secondo
- Da notare che l'incapsulamento non garantisce né la protezione dei dati nè nasconde i dettagli implementativi
- Position espone le proprietà `latitude` e `longitude` ai client
  - Client possono cambiare i dati senza nessun intervento da parte di `Position` → Incapsulamento non è sufficiente

Incapsulamento vs Information Hiding (18)

› Terza soluzione: SI incapsulamento, SI information hiding

```
public class Position {
  private double latitude;
  private double longitude;
  public Position(double latitude, double longitude) {
    setLatitude( latitude );
    setLongitude( longitude );
  }
  public void setLatitude(double latitude) {
    // Ensure -90 <= latitude <= 90 using modulo arithmetic.
    // Code not shown.
    // Then set instance variable.
    this.latitude = latitude;
  }
  ….
```

Incapsulamento vs Information Hiding (19)

```
…
  public void setLongitude(double longitude) {
    // Ensure -180 < longitude <= 180 using modulo arithmetic.
    // Code not shown.
    // Then set instance variable.
    this.longitude = longitude;
  }
  public double getLatitude() {
    return latitude;
  }
  public double getLongitude() {
    return longitude;
  }
…
```

Incapsulamento vs Information Hiding (20)

```
…
  public double distance(Position position) {
    // Calculate and return the distance from this object to the
       specified position.
    // Code not shown.
  }
  public double heading(Position position) {
    // Calculate and return the heading from this object to the
       specified position.
    // Code not shown.
  }
}
```

Incapsulamento vs Information Hiding (21)

```
Position myHouse = new Position(36.538611, -121.797500);

Position coffeeShop = new Position(36.539722, -121.907222);


double distance = myHouse.distance(coffeeShop);

double heading = myHouse.heading(coffeeShop);


System.out.println( "From my house at (" + myHouse.getLatitude() + ", "
+ myHouse.getLongitude() + ") to the coffee shop at (" +
coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() + ") is a
distance of " + distance + " at a heading of " + heading + "
degrees.");
```

Incapsulamento vs Information Hiding (22)

- La scelta di limitare i valori di `latitude` e `longitude` attraverso i metodi **setter** è strettamente una **decisione progettuale**
- L'incapsulamento **non** ha un ruolo
- Incapsulamento (in Java) **non** garantisce la protezione dei dati interni
- Come sviluppatore, sei libero di esporre i dati interni della tua classe
- E' necessario limitare l'accesso e la modifica degli elementi di dati interni mediante l'uso di metodi **getter** e **setter**

Incapsulamento vs Information Hiding (23)

› Link utili

https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

https://wiki.c2.com/?EncapsulationIsNotInformationHiding

https://www.javaworld.com/article/2075271/encapsulation-is-not-information-hiding.html

Costruttore (1)

- Viene utilizzato per creare un oggetto
- Sintassi

```
[ public |  private | protected ]  nomeClasse(lista parametri) {
      body
}
```

- Viene fornito un costruttore di default qualora non se ne dichiari uno
  - Non prende argomenti e non ha corpo (in realtà viene invocato `super()`)
  - Se è presente un costruttore quello di default **scompare**
- Esempio

```
public class Point {
      public Point(int x, int y) {
            …..
      }
}
```

Costruttore (2)

```
Point originOne = new Point(23, 94);

Rectangle rectOne = new Rectangle(originOne, 100, 200);

Rectangle rectTwo = new Rectangle(50, 100);
```

- Dichiarazione
  - E' necessario dichiarare una variabile con un tipo che rappresenta l'oggetto
- Istanziazione
  - Operatore `new` crea un nuovo oggetto
  - Viene allocato nell'heap dello spazio per contenere l'oggetto
  - Se non c'è spazio sufficiente viene lanciata eccezione `OutOfMemoryError`
  - Tutte le variabili di istanza vengono inizializzate al loro valore di default

Costruttore (3)

```
class Point {
   int x = 100;
   int y = 100;
   //Inizializzatore di istanza
   {
     System.out.println("x=" + x + ", y=" + y);
     System.out.println("Inizializzatore di istanza di Point");
     x = 200;
     y = 200;
   }
   Point() {
     System.out.println("x=" + x + ", y=" + y);
     System.out.println("Costruttore di default di Point");
     x = 300;
     y = 300;
   }
…
```

Costruttore (4)

```
…
    Point(int dx, int dy) {
      this();
      System.out.println("x=" + x + ", y=" + y);
      System.out.println("Costruttore con due parametri di Point");
      x = dx;
      y = dy;
    }
}
```

Costruttore (5)

- Inizializzazione
  1. Vengono assegnati i valori ai parametri formali del costruttore (se ce ne sono)
  2. Inizia l'esecuzione del costruttore
  3. Vengono eseguite le inizializzazioni esplicite delle variabili di istanza come appaiono nel codice sorgente
  4. Vengono eseguiti gli **inizializzatori** di istanza nell'ordine in cui appaiono nel codice sorgente
  5. Viene eseguito il corpo del costruttore

Costruttore (6)

- Valore di default per le variabili di istanza (valido anche per quelle di classe)
  - byte: `(byte)0`
  - short: `(short)0`
  - int: `0`
  - float: `0F`
  - double: `0`
  - char: `'\u0000'`
  - boolean: `false`
  - reference: `null`

Costruttore (7)

```
class Point {

   int x = 100;

   int y = 100;

   //Inizializzatore di istanza

   {

     System.out.println("x=" + x + ", y=" + y);

     System.out.println("Inizializzatore di
Point");

     x = 200;

     y = 200;

   }

   Point() {

     System.out.println("x=" + x + ", y=" + y);

     System.out.println("Costruttore di default
di Point");

     x = 300;

     y = 300;

   }
```

```
Point(int dx, int dy) {

     this();

     System.out.println("x=" + x + ", y=" + y);

     System.out.println("Costruttore con due
parametri di Point");

     x = dx;

     y = dy;

   }

}
```

Costruttore (8)

```
class ColoredPoint extends Point {

   int color = 0xFFFFFF; //Bianco

   //Inizializzatore di istanza

   {

            System.out.println("color=" + color);

            System.out.println("x=" + x + ", y=" + y);

            System.out.println("Inizializzatore
ColoredPoint");

            color = 0xFF00FF; //Magenta

   }

   public ColoredPoint(int x, int y, int c) {

         super(x, y);

         System.out.println("color=" + color);

         System.out.println("Costruttore di
ColoredPoint");

         color = c;

   }

}
```

```
class Test {

     public static void main(String[] args) {

        ColoredPoint cp = new

                  ColoredPoint(400, 500,

                           0xFF0000); //Rosso

        System.out.println("x=" + cp.x +

                  ", y=" + cp.y + ", color=" +

                           cp.color);

     }

}
```

Costruttore (9)

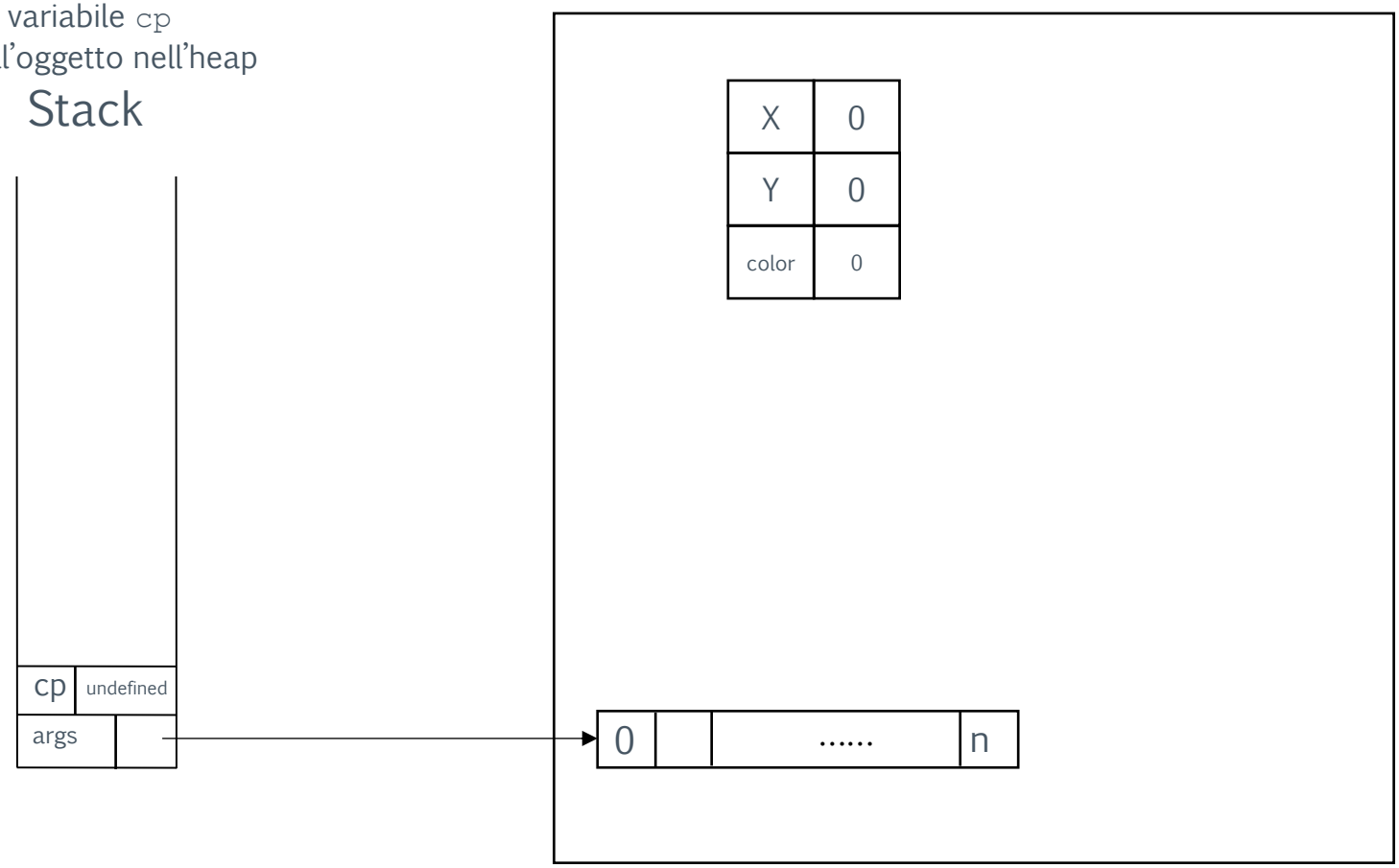
```
amleto@DESKTOP-P7TMPBB MINGW64 ~/Desktop/lezione
$ javac Test.java

amleto@DESKTOP-P7TMPBB MINGW64 ~/Desktop/lezione
$ java Test
x=100, y=100
Inizializzatore di istanza di Point
x=200, y=200
Costruttore di default di Point
x=300, y=300
Costruttore con due parametri di Point
color=16777215
x=400, y=500
Inizializzatore di istanza di ColoredPoint
color=16711935
Costruttore di ColoredPoint
x=400, y=500, color=16711680

amleto@DESKTOP-P7TMPBB MINGW64 ~/Desktop/lezione
$
```

Costruttore (10)

```
ColoredPoint cp = new ColoredPoint(400, 500, 0xFF0000);
```

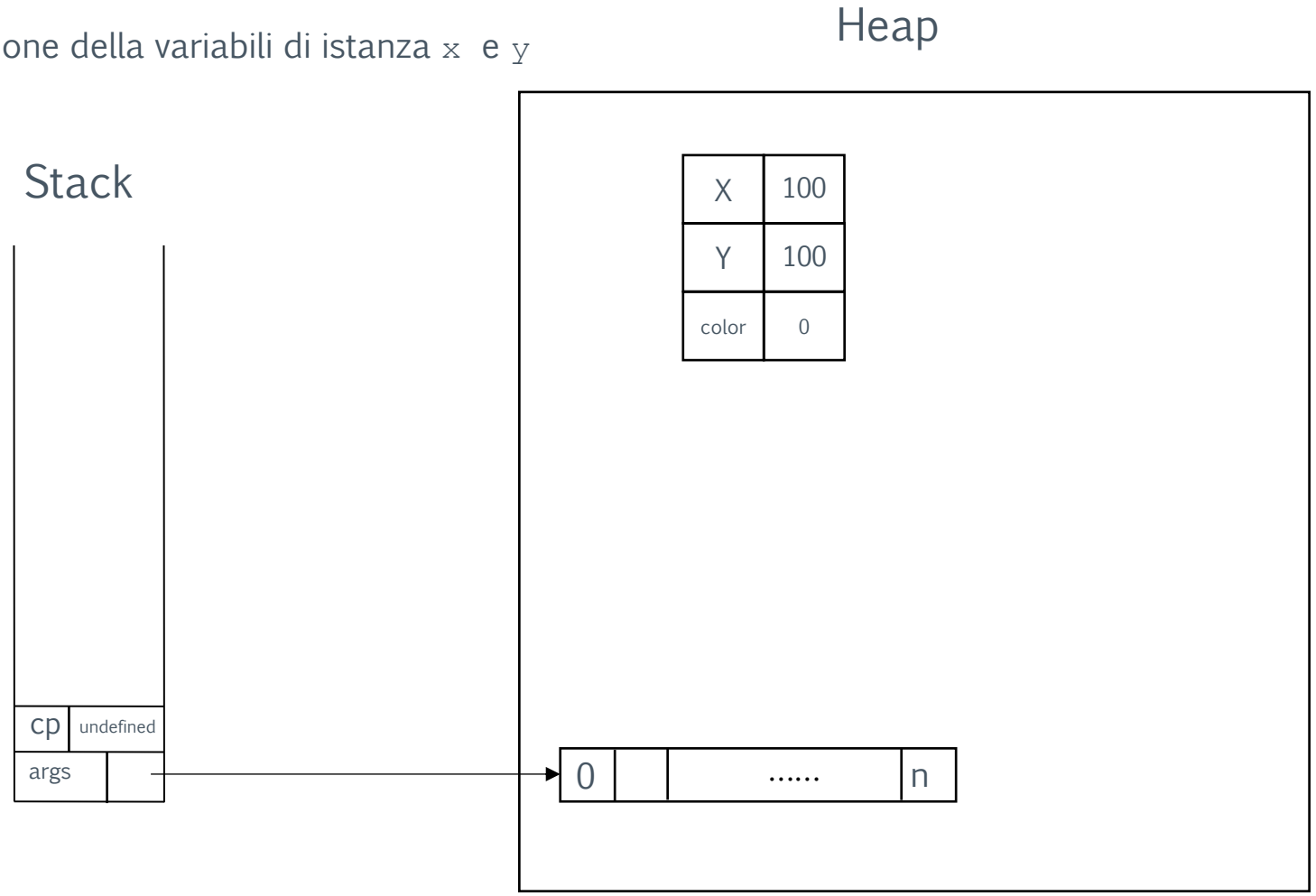
Heap

Dichiarazione variabile `cp`
Creazione dell'oggetto nell'heap



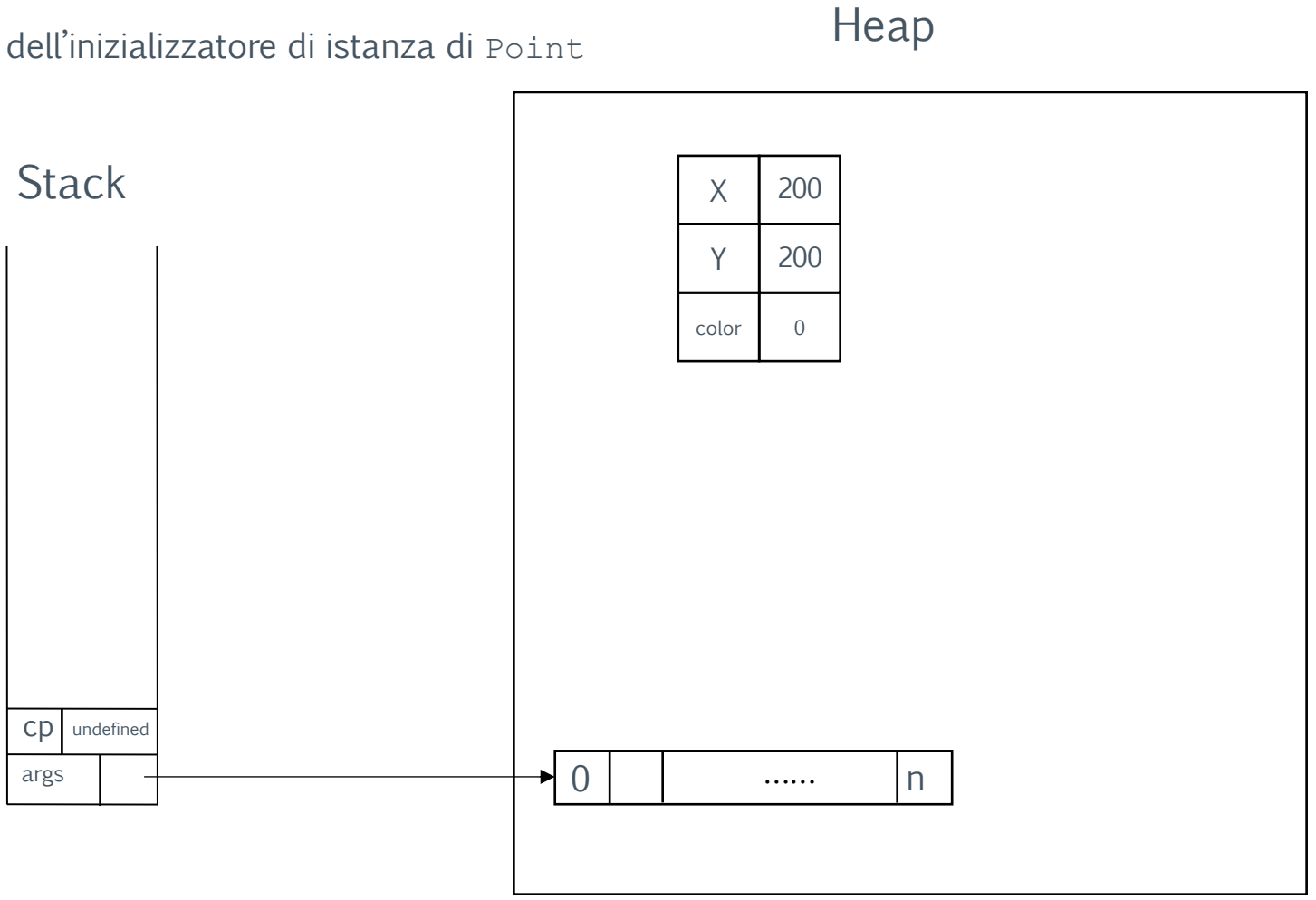
Costruttore (11)

Inizializzazione della variabili di istanza `x` e `y`



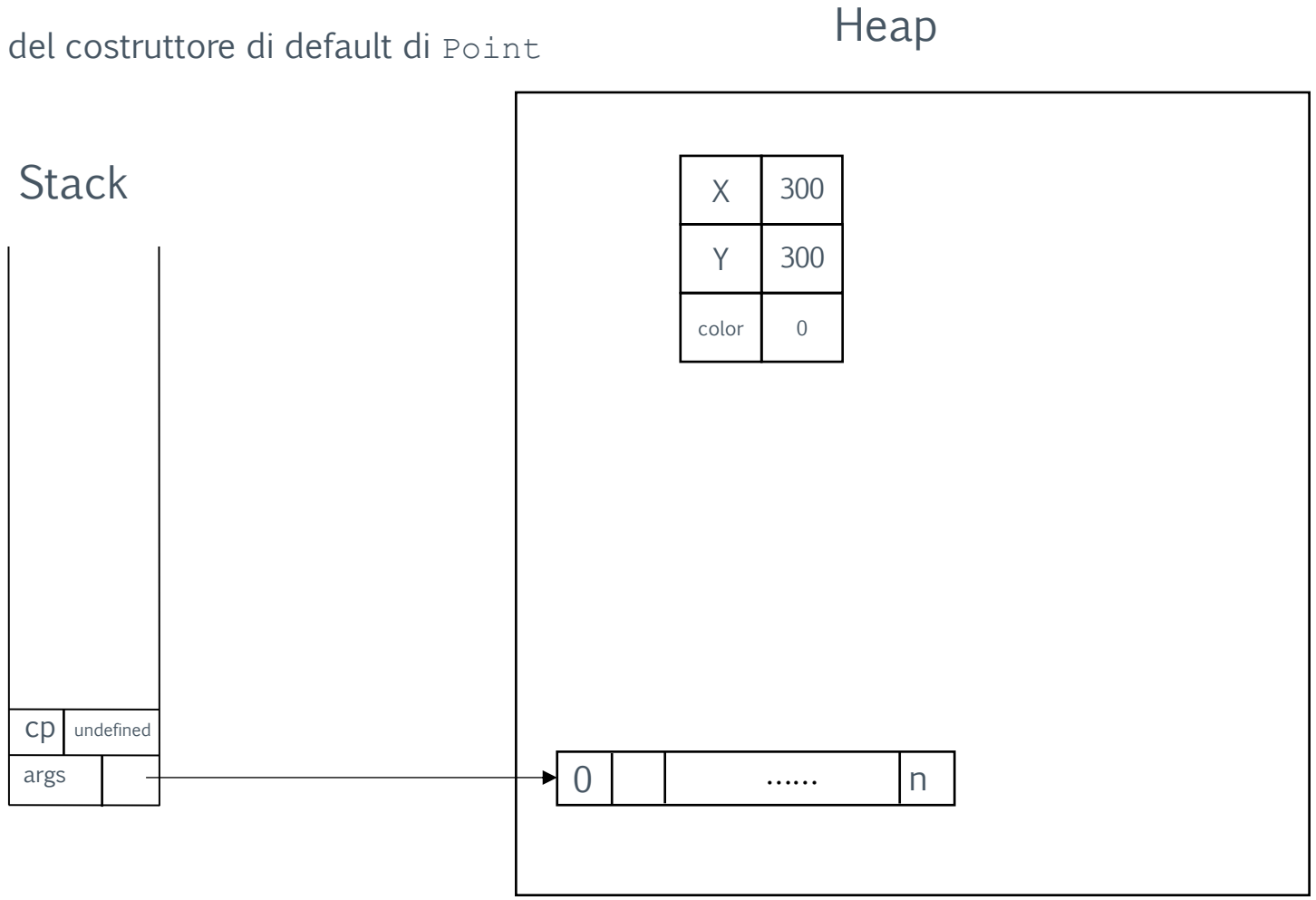
Costruttore (12)

Esecuzione dell'inizializzatore di istanza di `Point`



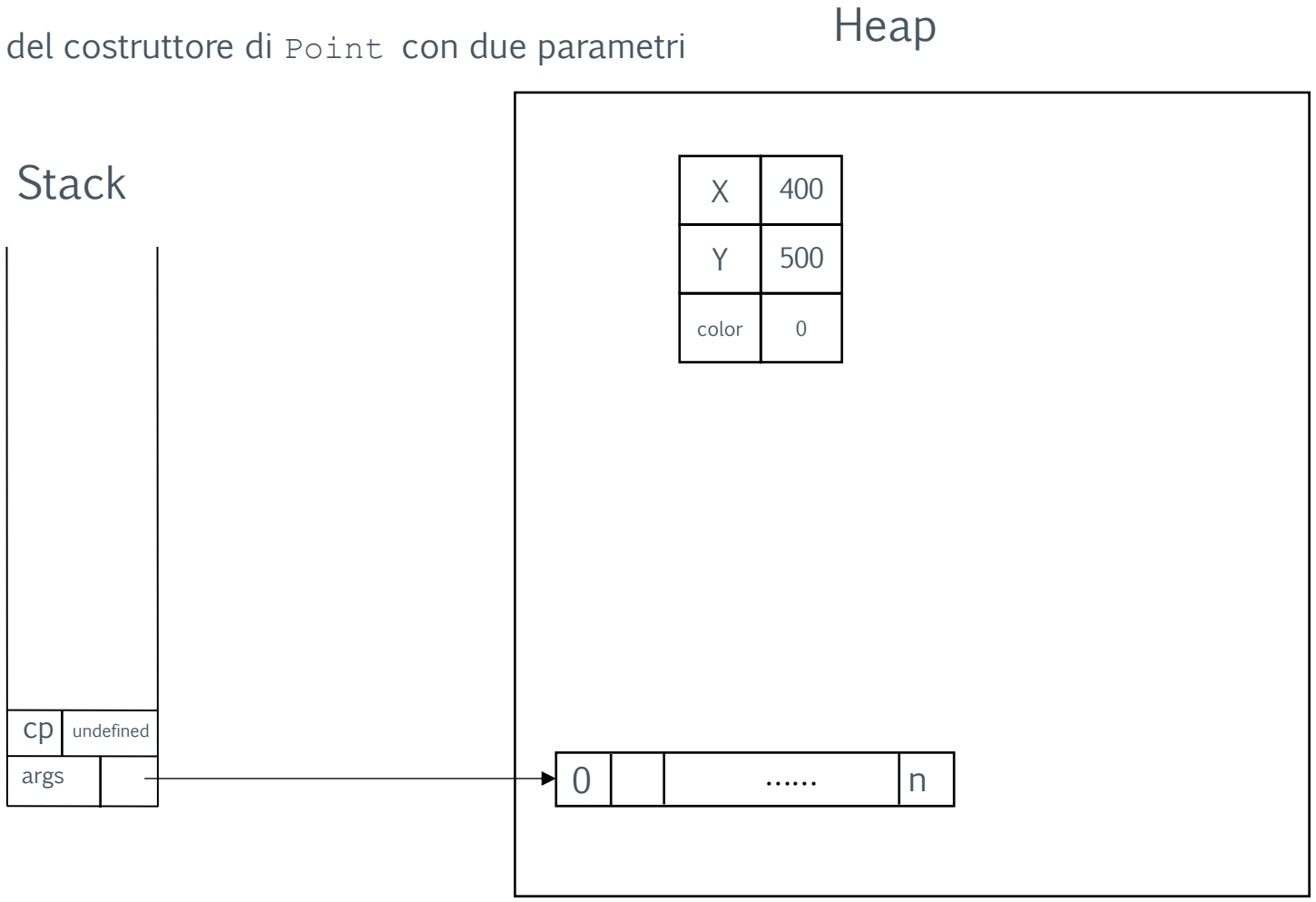
Costruttore (13)

Esecuzione del costruttore di default di `Point`



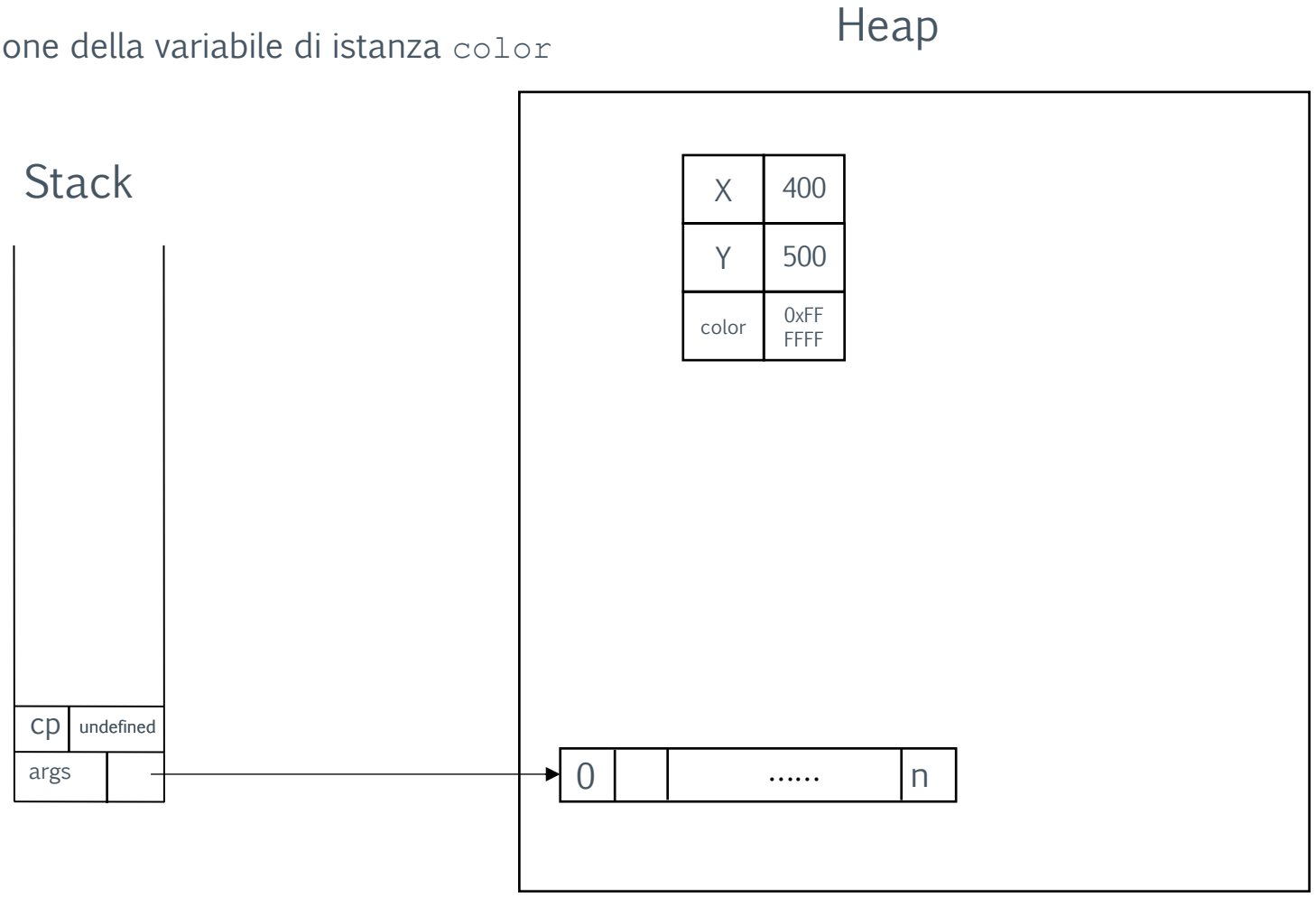
Costruttore (14)

Esecuzione del costruttore di `Point` con due parametri



Costruttore (15)

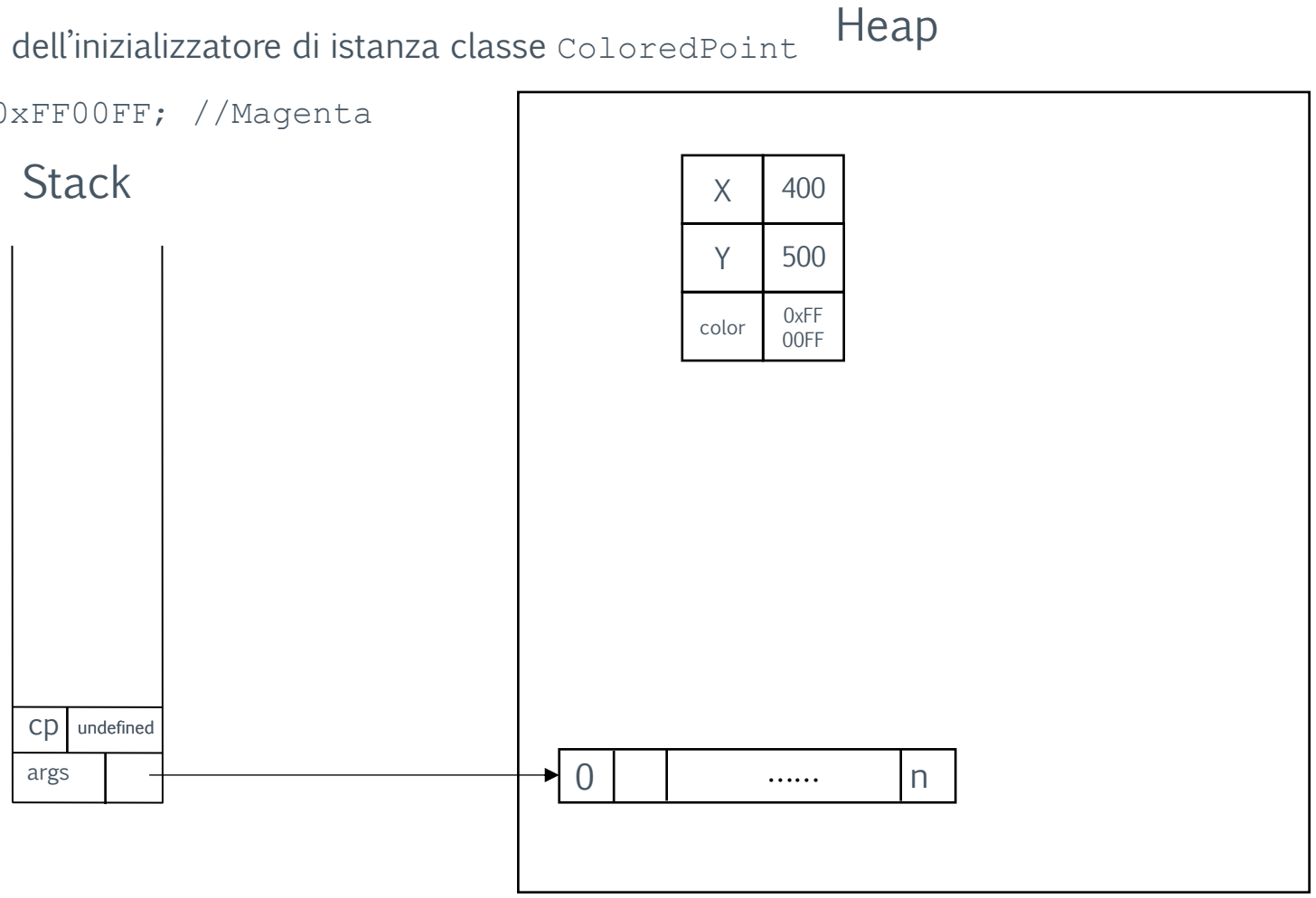
Inizializzazione della variabile di istanza `color`



Costruttore (16)

Esecuzione dell'inizializzatore di istanza classe `ColoredPoint`

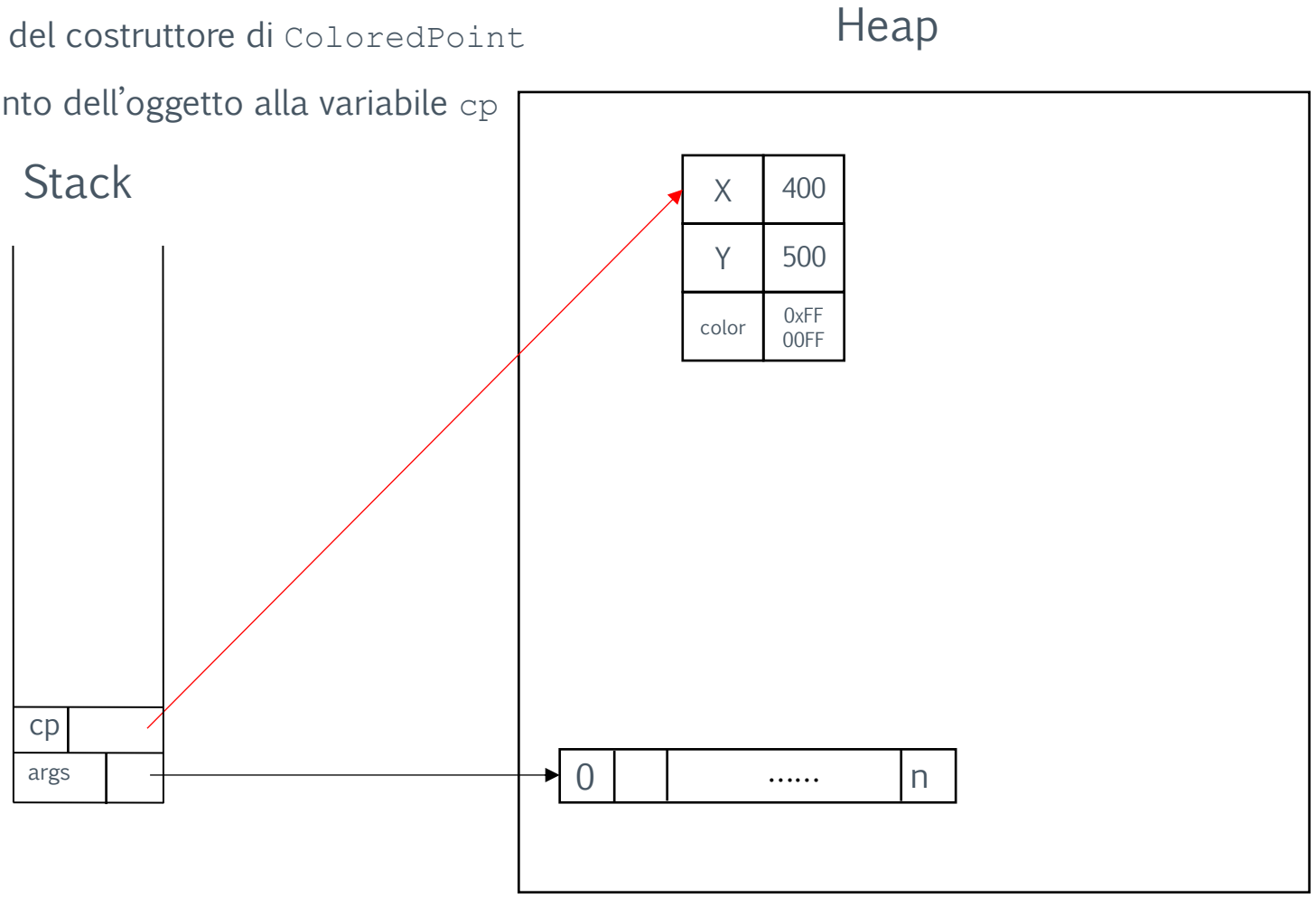
```
color = 0xFF00FF; //Magenta
```



Costruttore (17)

Esecuzione del costruttore di `ColoredPoint`

Assegnamento dell'oggetto alla variabile `cp`



Riferimento this (1)

- Indica il riferimento all'oggetto stesso
- Viene utilizzato nei seguenti ambiti
  - Per invocare all'interno di un costruttore un altro costruttore
  - All'interno di metodi e/o costruttori per **riferirsi** a variabili di istanza e/o metodi

Riferimento this (2)

```
public class Point {
   int x, y;
   public Point(int x, int y) {
          this.x = x;
          this.y = y;
  }
  public Point(Point p) {
          this.x = p.x;
          this.y = p.y;
  }
  public void move(int x, int y) {
          this.x = x;
          this.y = y;
  }
  public Point clonePoint() {
          Point p = new Point(this);
           return p;
  }
}
```

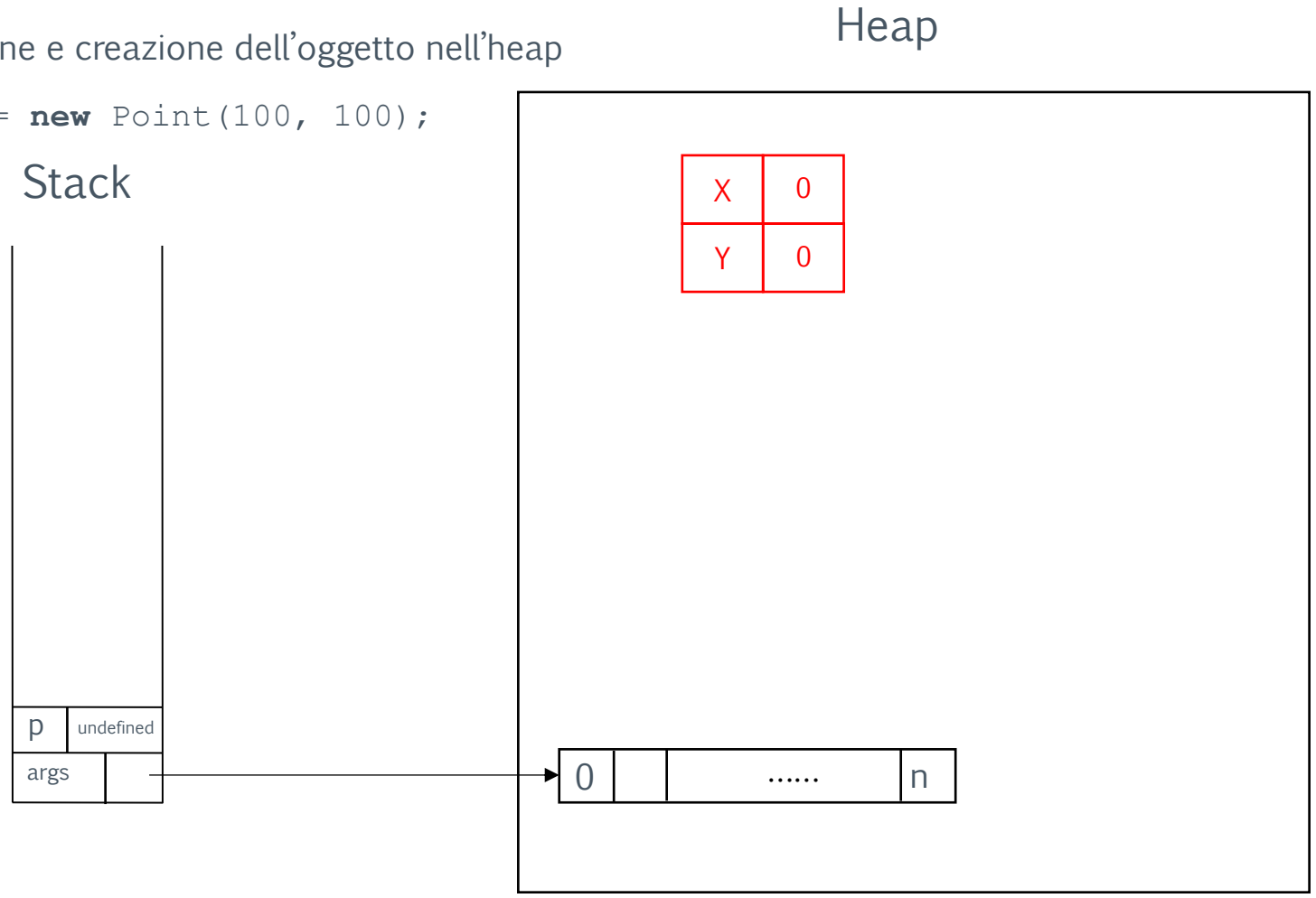
Riferimento this (3)

```
public class TestThis {
      public static void main(String[] args) {
            Point p = new Point(100, 100);
            Point p1 = p.clonePoint();
            p.move(200, 200);
            System.out.println("p.x: " + p.x);
            System.out.println("p.y: " + p.y);
            System.out.println("p1.x: " + p1.x);
            System.out.println("p1.y: " + p1.y);
      }
}
```

Riferimento this (4)

Dichiarazione e creazione dell'oggetto nell'heap

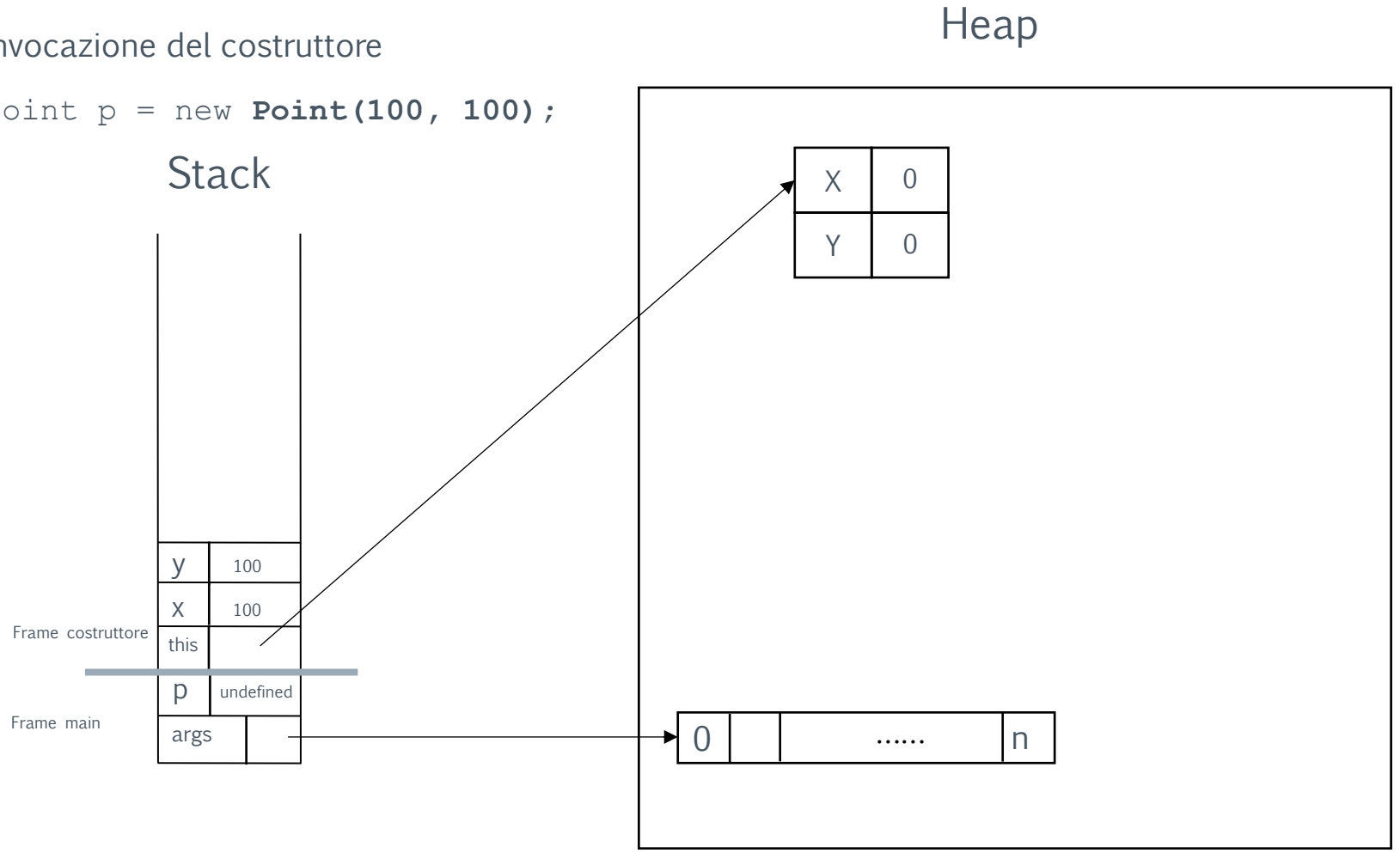
```
Point p = new Point(100, 100);
```



Riferimento this (5)

Invocazione del costruttore

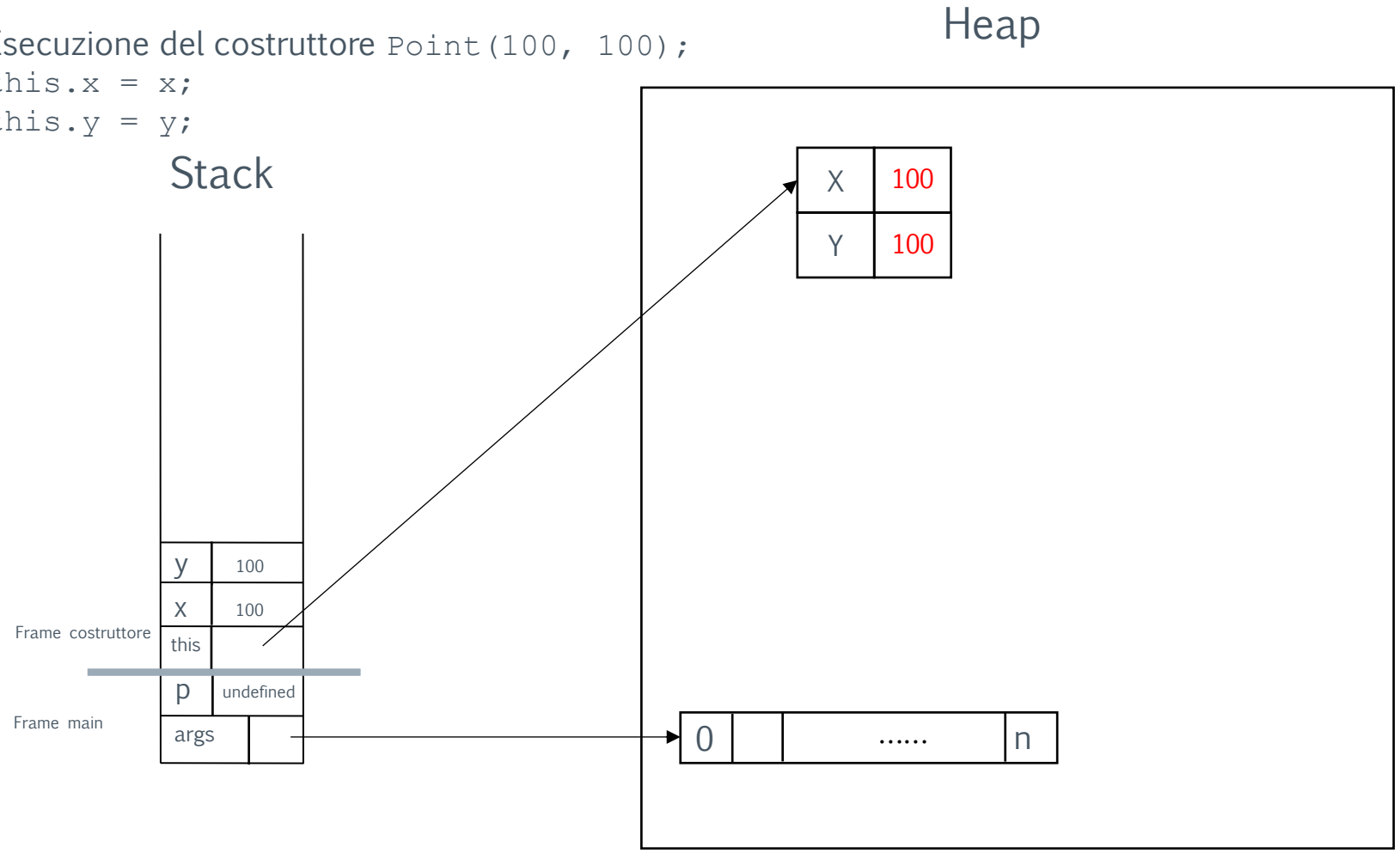
```
Point p = new Point(100, 100);
```



Riferimento this (6)

Esecuzione del costruttore `Point(100, 100);`

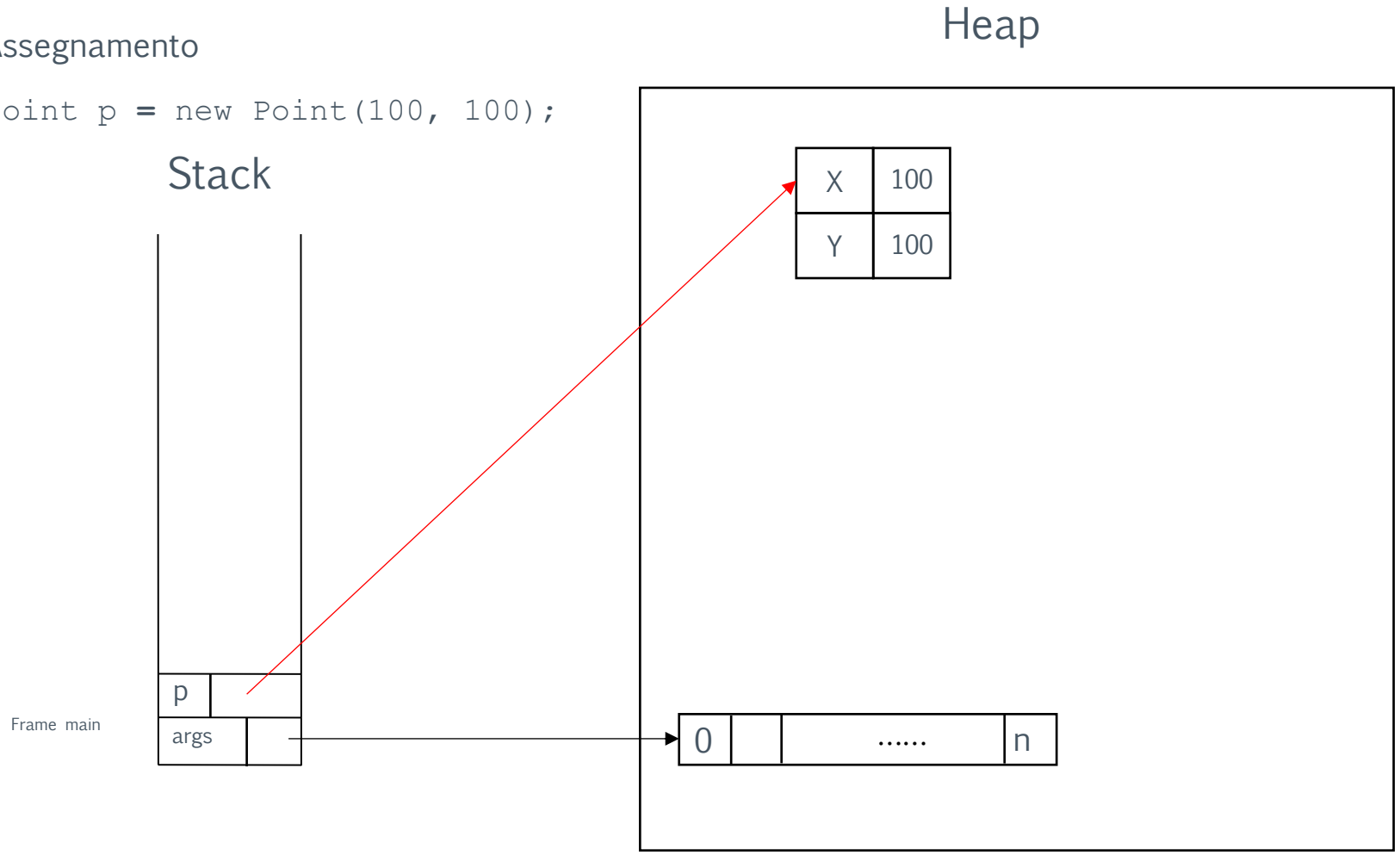
```
this.x = x;
this.y = y;
```



Riferimento this (7)

Assegnamento

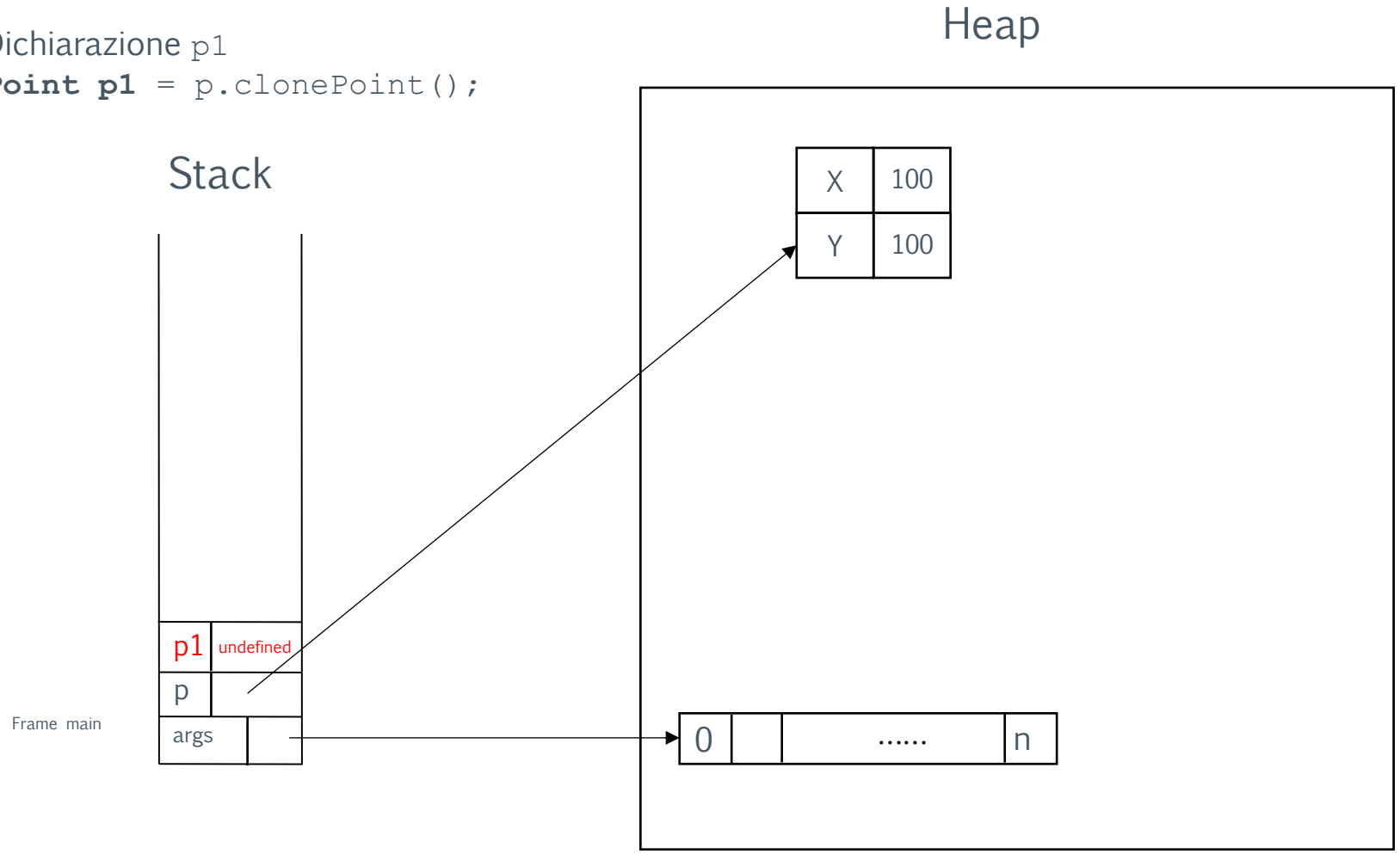
```
Point p = new Point(100, 100);
```



Riferimento this (8)

Dichiarazione `p1`

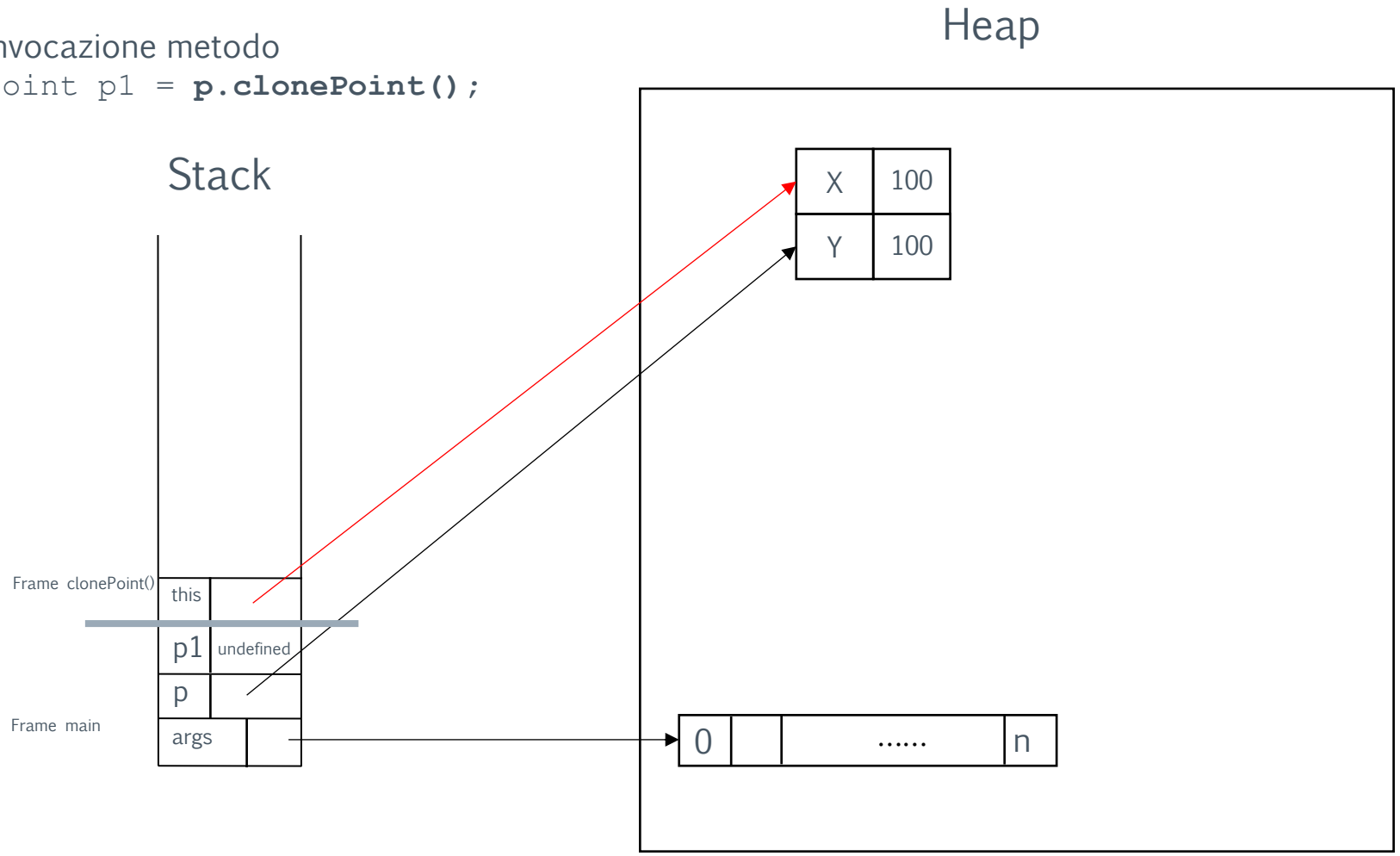
```
Point p1 = p.clonePoint();
```



Riferimento this (9)

Invocazione metodo

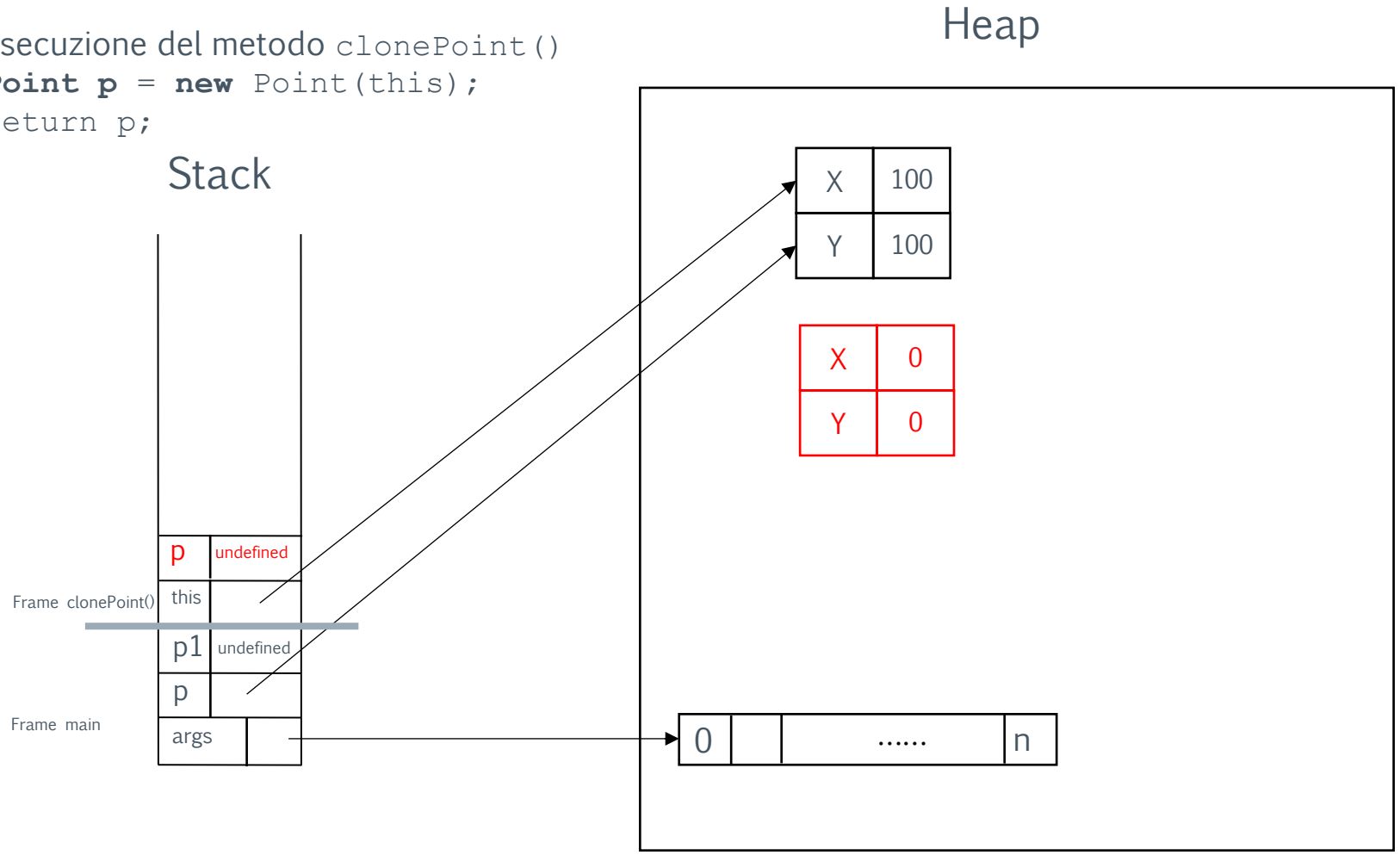
```
Point p1 = p.clonePoint();
```



Riferimento this (10)

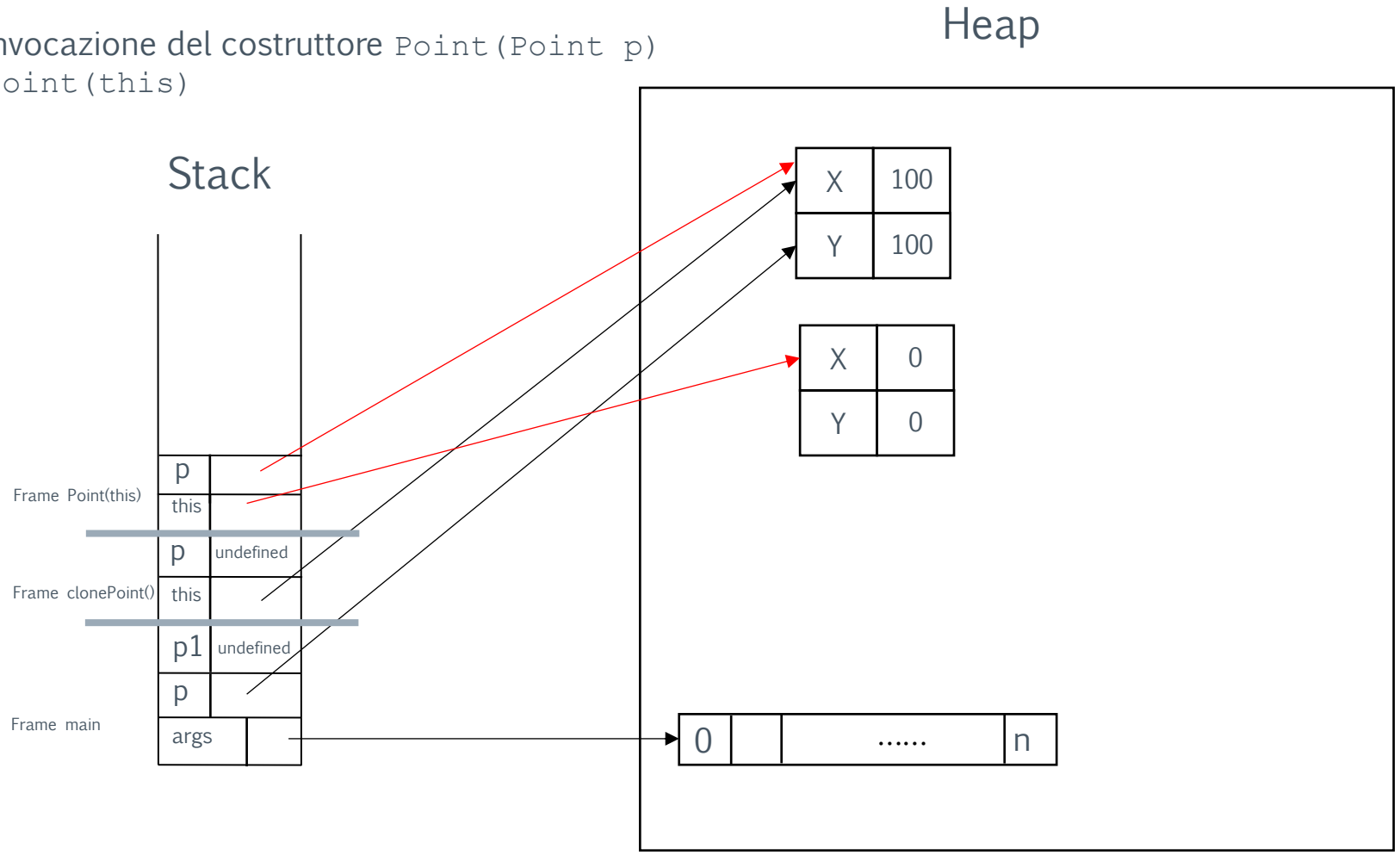
Esecuzione del metodo `clonePoint()`

```
Point p = new Point(this);
return p;
```



Riferimento this (11)

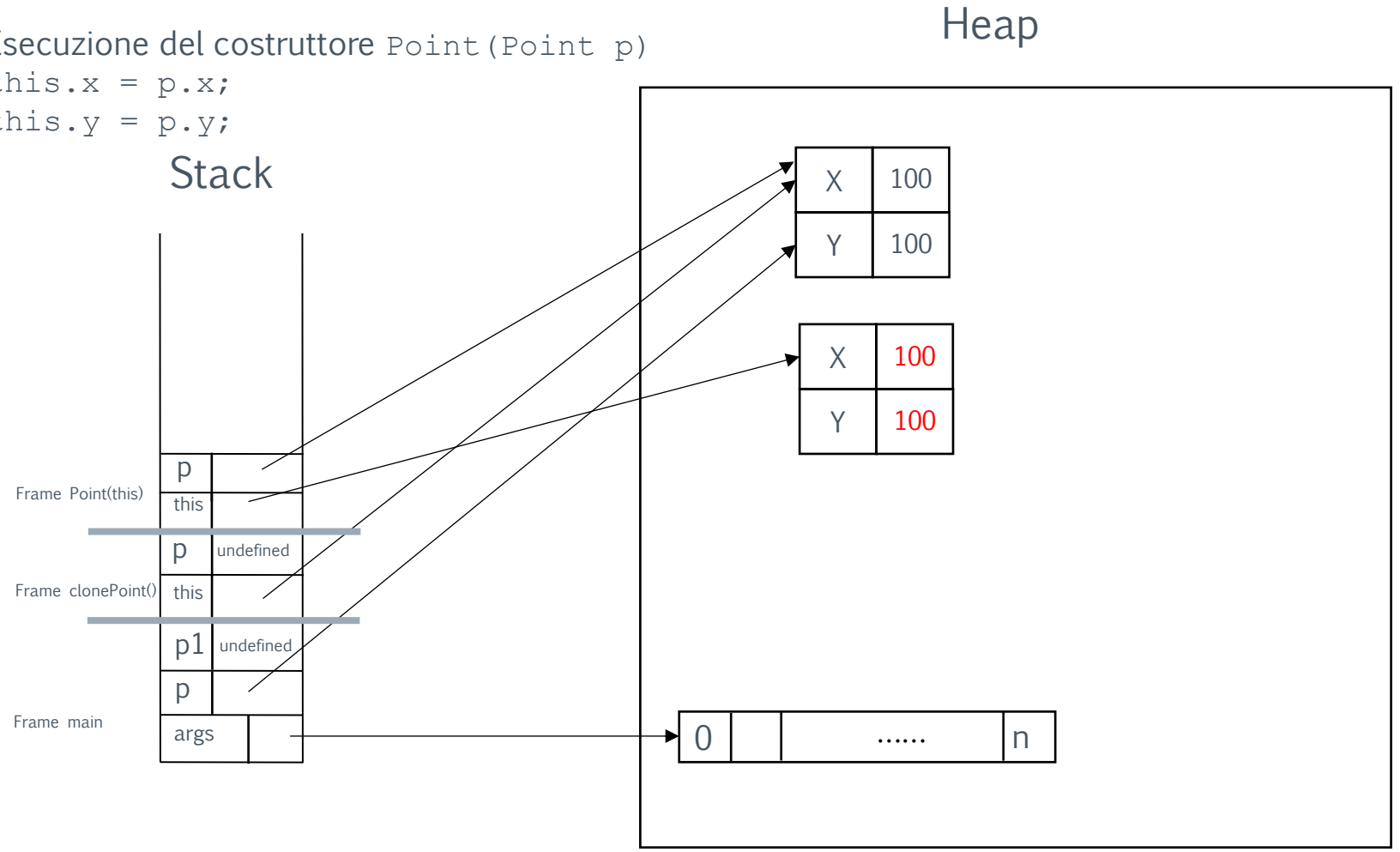
Invocazione del costruttore `Point(Point p)`
`Point(this)`



Riferimento this (12)

Esecuzione del costruttore `Point(Point p)`

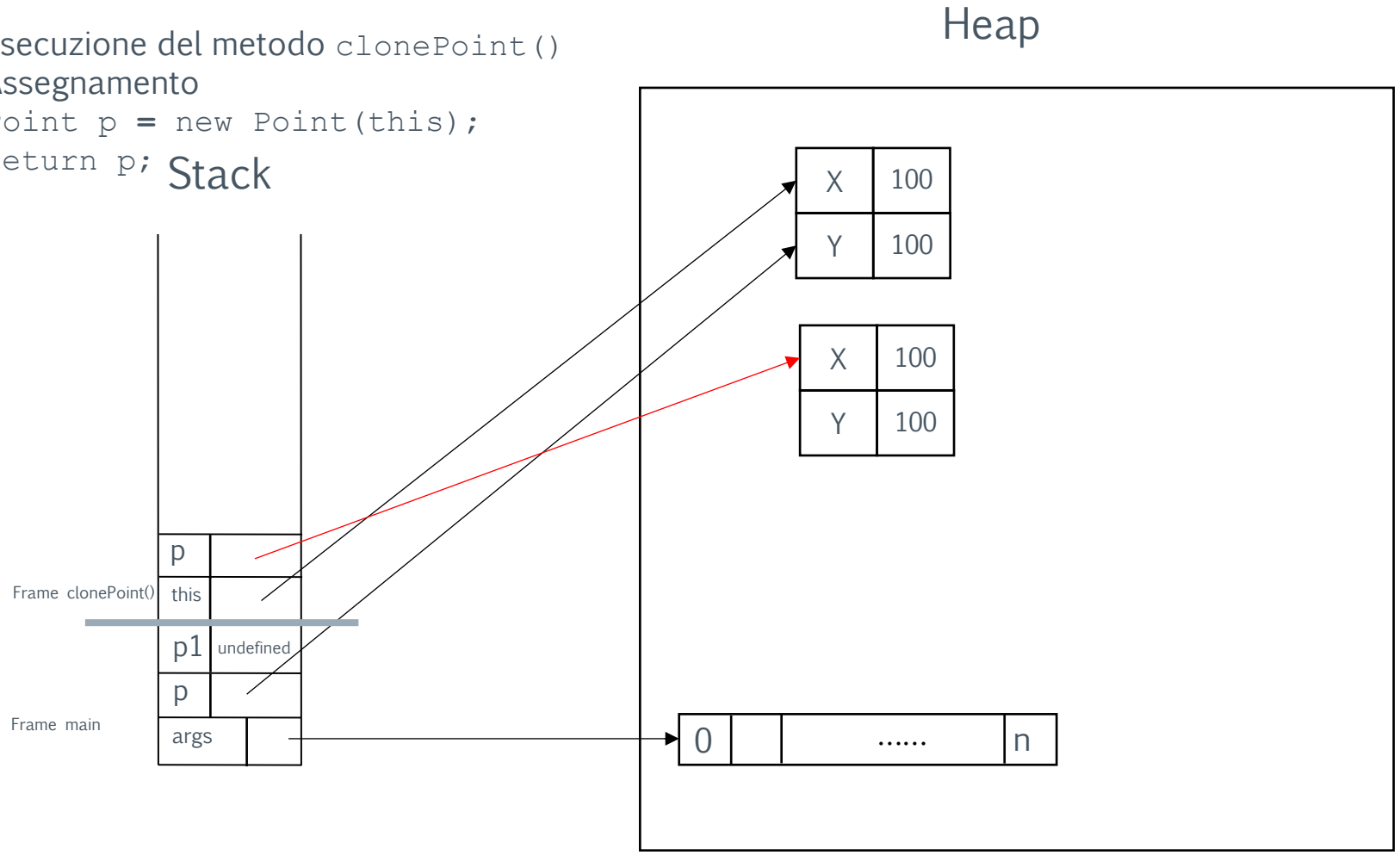
```
this.x = p.x;
this.y = p.y;
```



Riferimento this (13)

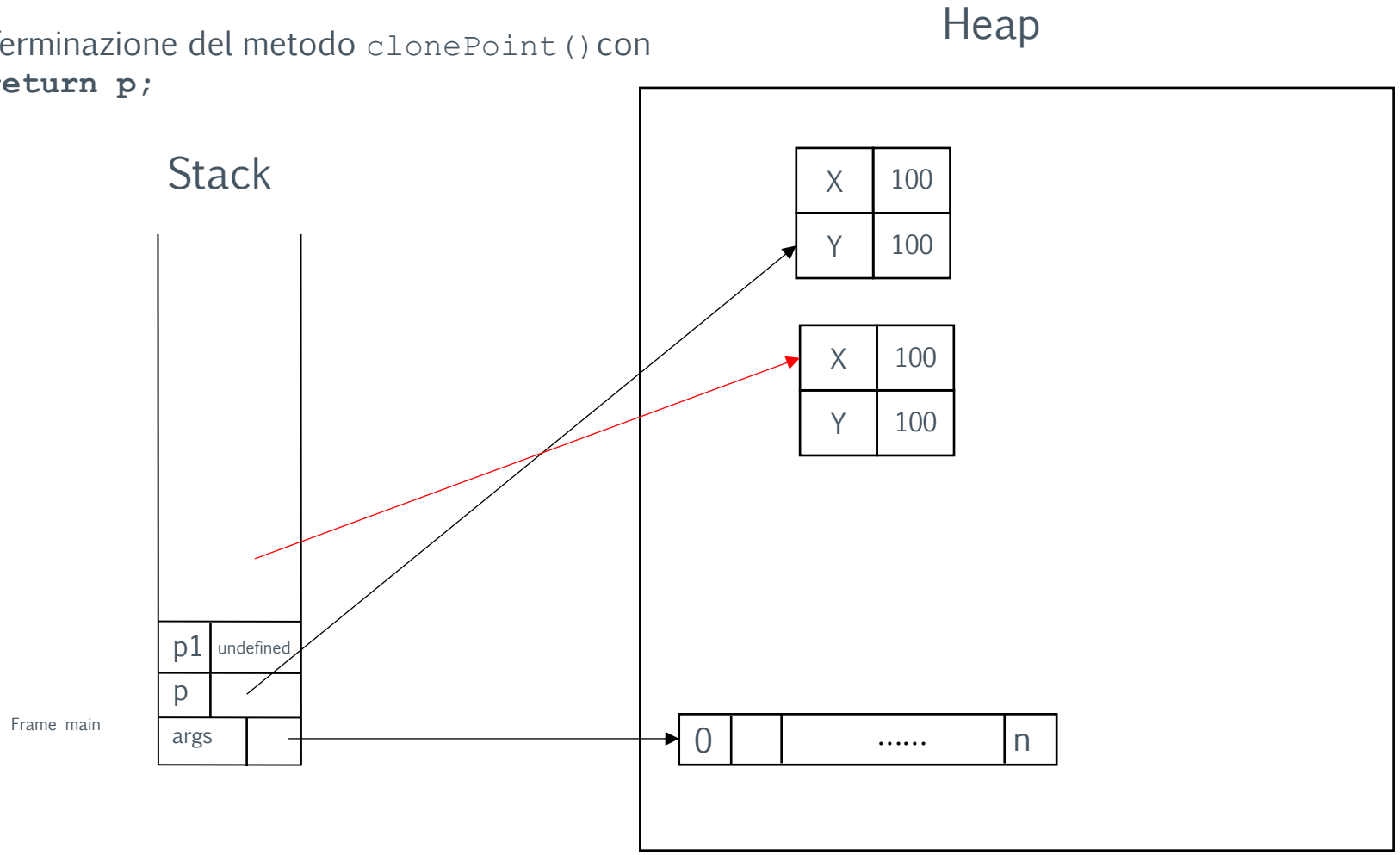
Esecuzione del metodo `clonePoint()`
Assegnamento

```
Point p = new Point(this);
return p;
```



Riferimento this (14)

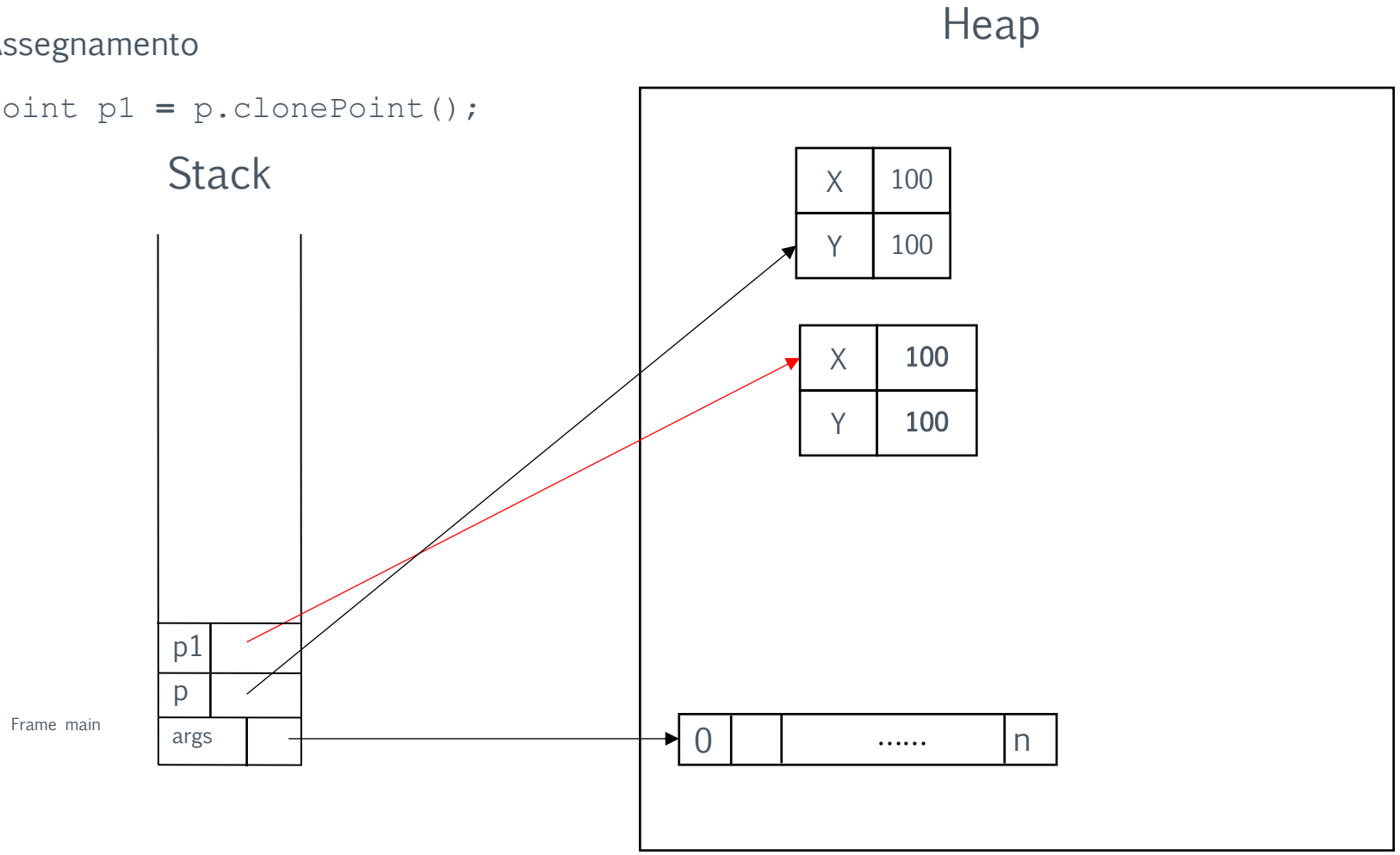
Terminazione del metodo `clonePoint()` con **`return p;`**



Riferimento this (15)

Assegnamento

```
Point p1 = p.clonePoint();
```



Riferimento this (16)

```
public class Cliente {
    private String nome;
    private String indirizzo;
    private String numeroDiTelefono;
    public Cliente (String nome, String indirizzo) {
        this(nome,indirizzo,"sconosciuto");
    }
    public Cliente(String nome, String indirizzo, String numeroDiTelefono) {
        this.setNome(nome);
        this.setIndirizzo(indirizzo);
        this.setNumeroDiTelefono( numeroDiTelefono);
    }
    public void setNumeroDiTelefono(String numeroDiTelefono) {
        numeroDiTelefono = numeroDiTelefono;
    }
…
```

Riferimento this (17)

```
    public void setIndirizzo(String indirizzo) {
        this.indirizzo = indirizzo;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getNome() {
        return nome;
    }
    public String getNumeroDiTelefono() {
        return numeroDiTelefono;
    }
    public String getIndirizzo() {
        return indirizzo;
    }
….
```

Riferimento this (18)

```
…
        public static void main(String[] args) {
                Cliente c = new Cliente("Juri Di Rocco", "Via Vetoio");
                c.setNumeroDiTelefono("0862/433735");
                System.out.println(c.getNome() + ": " + c.getIndirizzo() + ", " +
                                        c.getNumeroDiTelefono());
        }
}
```

Modificatori in JAVA

Access Modifiers	Non-Access Modifiers
private default or No Modifier protected public	static final abstract synchronized transient volatile strictfp

Variabili membro (1)

› Sintassi

```
[modificatoriAccesso] [static | final | transient | volatile]*
                                tipo     nome

modificatoriAccesso = [ public | private | protected ]
```

› Non possono apparire più di una volta

› Esempio

- `public static final String HELLO = "Hello";`

Variabili membro (2)

- `static`
  - Indica una variabile di classe
  - Vi è un **unico** valore associato alla classe indipendente da quante istanze della classe sono create
  - Si accede alla variabile mediante il nome della classe (`Math.PI`)
  - E' possibile accedere ad una variabile statica anche mediante una variabile (ad esempio locale) ma è **fortemente sconsigliato**
- Senza `static`
  - Indica una variabile di istanza
  - Vi sono **diversi valori** per ogni oggetto della classe

Variabili membro (3)

```
public class Point {
   private int x, y;
   public static int numPoints;

   public Point(int x, int y) {
      this.x = x;
      this.y = y;
      numPoints++;
   }
   public void move(int dx, int dy) {
      x = dx;
      y = dy;
   }
}
```

Variabili membro (4)

```
public class Test {

      public static void main(String[] args) {
            Point p1 = new Point(10,20);
            Point p2 = new Point(100,200);
            System.out.println(Point.numPoints++);
            System.out.println(p1.numPoints++);
            System.out.println(p2.numPoints);

      }
}
```

Ammesso ma fortemente sconsigliato

Variabili membro (5)

Inizializzatore statico

- E' un blocco di codice che viene eseguito **una volta soltanto** quando la classe viene caricata in memoria
- Generalmente viene utilizzato per inizializzare **variabili di classe**
- E' possibile inserire in una classe più inizializzatori statici; Verranno eseguiti dall'alto verso il basso
- Sintassi

```
static {
      < statements>
}
```

Variabili membro (6)

```
public class EsempioStatico {
    private static int a;
    public EsempioStatico(){
        a += 10;
    }
    static {
        System.out.println("valore statico = " + a);
        a = 10;
        System.out.println("valore statico = " + a);
    }
    public static void main(String[] args) {
      System.out.println("valore statico = " + a);
      EsempioStatico e = new EsempioStatico();
      System.out.println("valore statico = " + a);
    }
}
```

Variabili membro (7): final

- Indica una variabile che può essere assegnata una **sola volta** ovvero costante
- Può essere applicata a **variabile di istanza, di classe (statica), locale, parametro di un metodo**
- Una volta assegnata non può essere più modificata
  - Tipo reference non può essere modificato il **riferimento.** Valori all'interno dell'oggetto possono essere modificati?
- Generalmente le costanti scritte in maiuscolo
- Generalmente vengono utilizzate in congiunzione con parola chiave `static`
- `const` parola chiave riservata ma non utilizzata

Variabili membro (7): final

- Indica una variabile che può essere assegnata una **sola volta** ovvero costante
- Può essere applicata a **variabile di istanza, di classe (statica), locale, parametro di un metodo**
- Una volta assegnata non può essere più modificata
  - Tipo reference non può essere modificato il **riferimento.** Valori all'interno dell'oggetto possono essere modificati?
  - OVVIAMENTE SI
- Generalmente le costanti scritte in maiuscolo
- Generalmente vengono utilizzate in congiunzione con parola chiave `static`
- `const` parola chiave riservata ma non utilizzata

Variabili membro (8): final

- E' possibile assegnare il valore della variabile **NON** contestualmente alla dichiarazione (*blank final*)
  - Variabile di istanza
    - E' necessario inizializzarla in **tutti** i costruttori oppure all'interno di un inizializzatore di istanza
  - Variabile di classe
    - E' necessario inizializzarla all'interno di un inizializzatore statico

Variabili membro (9): final

```
public class Constants {

  public static void main(String[] args) {
            final double CM_PER_INCH = 2.54;
            System.out.println("Centimetri per inch:" + CM_PER_INCH);
            CM_PER_INCH = 10.30;  //???
            Constants.CM_PER_INCH = 20; //???
      }

      public static final double CM_PER_INCH = 2.54;

}
```

Variabili membro (9): final

```
public class Constants {

  public static void main(String[] args) {
            final double CM_PER_INCH = 2.54;
            System.out.println("Centimetri per inch:" + CM_PER_INCH);
            CM_PER_INCH = 10.30;  //ERRORE IN COMPILAZIONE
            Constants.CM_PER_INCH = 20; //ERRORE IN COMPILAZIONE
      }

      public static final double CM_PER_INCH = 2.54;

}
```

Variabili membro (10): final

```
class Point {
        int x,y;
        final Point root;
        static final Point ORIGIN;
        static {
                ORIGIN = new Point(0,0);
        }
        public Point(int x, int y){
                this.x = x;
                this.y = y;
                root = null;
        }
        public Point(int x, int y, Point root){
                this.x = x;
                this.y = y;
                this.root = root;
        }
}
```

Variabili membro (11): final

```
class Test {

   public static void main(String[] args) {
      Point p1 = new Point(100, 100);
      Point p2 = new Point(200, 200, p1);

      p1.root = new Point(300, 300); //ERRORE IN COMPILAZIONE
      p2.root.x = 300;               //OK
      System.out.println("ORIGIN.x:" + Point.ORIGIN.x);

      Point.ORIGIN = new Point(1000, 1000); //ERRORE IN COMPILAZIONE
      Point.ORIGIN.x = 400;                 //OK
   }
}
```

Metodi (1)

› Sintassi

```
[modificatoriAccesso]
            [ static | abstract | final | native | synchronized ]*
               tipoRitorno nomeMetodo(listaparametri)
                                  [ throws exceptions]

modificatoriAccesso = [ public |  private | protected ]
```

› Non possono apparire più di una volta

› Esempio

```
public static final String getHello() {
      return “Hello World”;
}
```

Metodi (2)

- La **Segnatura** di un metodo (o costruttore) è composta da
  - Nome
  - Numero e tipi dei parametri formali
  - Non fanno parte della segnatura il tipo di ritorno e la clausola delle eccezioni
- Non è possibile avere **due** metodi con la stessa segnatura all'interno di una classe (es.: stessi metodi con diverso tipo di ritorno)
- Java 5 ha leggermente modificato tale concetto per i generics, ma è sostanzialmente valido
- Esempio
  - `public void move(int x, int y)`
  - `public int move(int dx, int dy)`

Metodi (3)

- `static`
  - Indica un metodo di classe (o statico)
  - E' associato alla classe e può essere invocato senza un particolare riferimento ad un oggetto
  - Non è possibile utilizzare `this` o `super` e variabili di istanza
  - Non può essere dichiarato `abstract`
- Senza `static`
  - Indica un metodo di istanza
  - E' invocato rispetto ad un oggetto
  - Può utilizzare/modificare variabili di istanza

Metodi (4)

```
public class Point {
   private int x, y;
   private static int numPoints;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
      numPoints++;
   }
   public void move(int dx, int dy) {
      x = dx;
      y = dy;
   }
   public static int getNumPoints() {
      return numPoints;
   }
}
```

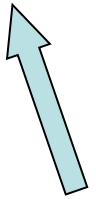
Package (1)

- Collezione di *classi* ed *interfacce* in relazione tra di loro che fornisce una protezione all'accesso e alla gestione del namespace
- Classi e interfacce che fanno parte della piattaforma Java sono raggruppati in package
- Pacchetti standard di java sono organizzati in modo *gerarchico*
- Livello più alto `java` e `javax`
- Esempi
  - `java.lang, java.io, java.net, java.awt, java.util`
  - `javax.swing, javax.swing.border`

Package (2)

- Package garantiscono **l'univocità** dei nomi delle classe e interfacce
- **Nome completo** di una classe ed interfaccia
  - `it.univaq.disim.java.Employee`
  - `com.horstmann.corejava.Employee`
- Sintassi

```
[ package <nome_top_package>[.<nome_sub_package>]*; ]
<dichiarazione classe e/o interfaccia>
```



Non dichiarato pacchetto di default (o senza nome)

Package (3)

- E' necessario specificare la dichiarazione del `package` all'inizio del file sorgente
- Non è possibile avere più dichiarazioni `package` in un file sorgente
- Se **non** è presente la dichiarazione del package allora la classe/interfaccia appartiene al package di default (o senza nome)

Package (4)

› Esempio

```
package graphics;
public class Circle extends Graphic implements Draggable {
. . .
}
```

Nome completo: `graphics.Circle`

```
package it.univaq.disim.java.graphics;
public class Circle extends Graphic implements Draggable {
. . .
}
```

› **Nome completo**: `it.univaq.disim.java.graphics.Circle`

Package (5)

- Una classe può utilizzare tutte le classi contenute nello stesso pacchetto e tutte le classi/interfacce *pubbliche* di altri pacchetti
- E' possibile accedere alle classi pubbliche in due modi
  - Nome completo
  - Nome semplice + con keyword `import`
- `java.lang` sono importati per default
- Esempio

```
package graphics;
class Rectangle extends java.awt.Component {
…
}
```

Package (5)

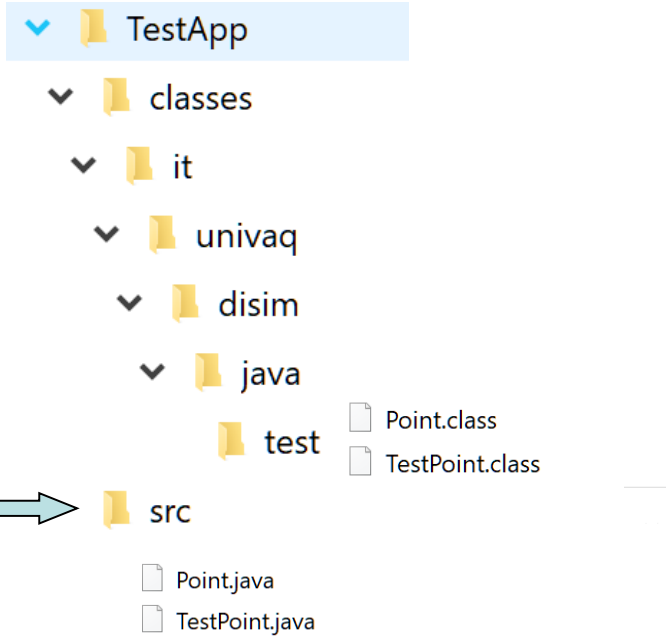
› Esempio 2

```
package graphics;

import java.awt.Component;

class Rectangle extends Component {

…

}
```

Layout sorgenti e destinazione (1)

- Se una classe è dichiarata all'interno di un package **non** è necessario che i sorgenti si trovino all'interno di una directory con lo stesso nome del package
- E' valido anche se il package è composto (es. `it.univaq.disim.java.testpackage`)
- La struttura deve invece essere mantenuta per le classi compilate

Layout sorgenti e destinazione (2)



Layout sorgenti e destinazione (3)

```
package it.univaq.disim.java.test;

public class Point {

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

     public String toString() {
            return "x= " + x + ", y= " + y;
     }

}
```

Layout sorgenti e destinazione (4)

```
package it.univaq.disim.java.test;

public class TestPoint {

     public static void main(String[] args) {
          Point p = new Point(10, 20);
          System.out.println("Point " + p );
     }
}
```

Import statico (1)

- Per accedere a variabili o metodi statici si utilizza il nome della classe (semplice o completo)
- Esempio

```
public class Test {


	public static void main(String[] args) {
		double pigreco = Math.PI;
		System.out.println(pigreco);
	}


}
```

Import statico (2)

```
import static java.lang.Math.*;
public class Test {

      public static void main(String[] args) {
            double pigreco = PI;
            System.out.println(pigreco);
      }

}
```

Se ne sconsiglia l'utilizzo

Modificatori di accesso

Modificatore	Classe	Sottoclasse	Package	mondo
`private`	X			
`protected`	X	X*	X	
`public`	X	X	X	X
`package`	X		X	

Permette di controllare **l'accesso alle variabili** e ai metodi rispetto alla classe, sottoclassi, ai membri dello stesso package e al resto del *mondo*

package: non è l'utilizzo della keyword ma è quando non si mette alcun modificatore

Permettono di realizzare la cosiddetta *information hiding*

Modificatori di accesso: public (1)

- E' il più semplice dei modificatori
- Chiunque può accedere alle variabili, metodi e costruttori dichiarati pubblici
- E' possibile utilizzare `public` anche nella definizione di una classe → Indica che tale classe è visibile al di fuori del package

Modificatori di accesso: public (2)

```
package greek;
public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}

package roman;
import greek.*;
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;       // legale
        a.publicMethod();       // legale
    }
}
```

Modificatori di accesso: private (1)

- Più restrittivo è `private`
- Può accedere ad un membro privato (variabile o metodo) soltanto la classe dove è definito
- Generalmente si dichiarano variabili o metodi privati che renderebbero l'oggetto in uno stato inconsistente
- Da notare che l'accesso è **riservato alla classe e non all'oggetto**
- E' possibile utilizzare tale modificatore anche in un **costruttore**

Modificatori di accesso: private (2)

```
class Alpha {
    private int iamprivate;
    private void privateMethod() {
        System.out.println("privateMethod");
    }
}

class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprivate = 10;      // illegal
        a.privateMethod();      // illegal
    }
}
```

Modificatori di accesso: private (3)

```
class Alpha {
    private int iamprivate;
    boolean isEqualTo(Alpha anotherAlpha) {
        if (this.iamprivate == anotherAlpha.iamprivate) //Accesso relativo alla classe
            return true;
        else
            return false;
    }
}

class Utility {
   private Utility() {
   }
}
Utility u = new Utility(); //Errore in compilazione
```

Modificatori di accesso: protected (1)

- Può accedere ad un membro protetto (**variabile, metodo o costruttore**) soltanto
  - Classe dove è definito
  - Sottoclassi (con qualche eccezione)
  - Classi che appartengono allo stesso package
- Generalmente è utilizzato quando si ha bisogno di far accedere ai membri alle sottoclassi e non a classi che non sono in relazione

Modificatori di accesso: protected (2)

```
public class Point {
   protected int x,y;
}

public class Point3D extends Point {
   protected int z;

   public void move(int x, int y, int z) {
            this.x = x;                  //OK
            this.y = y;                  //OK
            this.z = z;                  //OK
   }
}
```

Modificatori di accesso: protected (3)

```
package greek;
public class Alpha {
    protected int iamprotected;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}

package greek;
class Gamma {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprotected = 10;    // legale
        a.protectedMethod();    // legale
    }
}
```

Modificatori di accesso: protected (4)

```
package latin;
import greek.*;

class Delta extends Alpha {
    void accessMethod(Alpha a, Delta d) {
       a.iamprotected = 10;    // errore in compilazione
       d.iamprotected = 10;    // legale
       a.protectedMethod();    // errore in compilazione
       d.protectedMethod();    // legale
    }
}
```

Modificatori di accesso: package (1)

- Definito quando non si specifica **nessun** modificatore di accesso
- Livello di accesso è ammesso alle classi dello stesso package
- E' possibile *utilizzarlo* anche nella definizione di una classe → classe è visibile all'interno del package

Modificatori di accesso: package (2)

```
package greek;
class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}

package greek;
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;     // legale
        a.packageMethod();     // legale
    }
}
```