

04

Oggetti e Classi pt.2

inizializzazione, accessi, distruzione

Mirko Viroli
`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2022/2023

Goal della lezione

- Completare i meccanismi OO di Java a livello di classe
- Porre le basi per poter fare della corretta progettazione
- Discutere aspetti collegati alla vita di un oggetto (costruzione, accesso, distruzione)

Argomenti

- Codice statico
- Costruttori
- Overloading di metodi
- Approfondimento sui package
- Controllo d'accesso
- Finalizzazione e garbage collection

- 1 Codice statico
- 2 Costruttori
- 3 Controllo d'accesso
- 4 Distruzione oggetti
- 5 Una applicazione

Codice statico

Meccanismo

- Alcuni campi e metodi di una classe possono essere dichiarati **static**
- Esempi visti:
 - ▶ il `main` dal quale parte un programma Java
 - ▶ il metodo `java.util.Arrays.toString()` (ossia, il metodo `toString` nella classe `Arrays` dentro al package `java.util`)
 - ▶ il campo `java.lang.System.out`
 - ▶ il metodo `java.lang.System.getProperty()`
- Tali campi e metodi sono considerati proprietà della classe, o comunque dell'insieme di tutti gli oggetti, non di un singolo oggetto
- Sintassi: `classe.metodo(argomenti)`, e `classe.campo`
 - ▶ omettendo la classe si assume quella da cui parte la chiamata

Sulla notazione

- Iniziano con minuscolo: nomi di package, metodi e campi
 - Iniziano con maiuscolo: nomi di classe
- ⇒ Questo consente di capire facilmente il significato delle istruzioni

Uso di codice statico o non statico in una classe?

Codice non statico (detto anche codice istanza)

- È codice puro object-oriented
- Definisce le operazioni e lo stato di ogni oggetto creato dalla classe

Codice statico

- Non è codice puro object-oriented, ma codice in stile “imperativo/strutturato”
- Definisce funzioni e variabili del software definito dalla classe
 - ▶ possono essere visti come metodi e campi della classe (non dei suoi oggetti)

Usarli assieme?

- Le helper class abbiano solo metodi/campi statici
- Le classi OO non abbiano metodi/campi statici, a meno di qualche funzionalità “generale” della classe (si veda nel prosieguo)

Esempio Point3D

```
1 class Point3D { // dichiarazione classe
2     double x; // 3 campi
3     double y;
4     double z;
5
6     void build(double a, double b, double c) {
7         this.x = a;
8         this.y = b;
9         this.z = c;
10    }
11
12    double getSquaredModulus() {
13        return this.x * this.x + this.y * this.y + this.z * this.z;
14    }
15
16    static Point3D zero = new Point3D(); // 0,0,0
17
18    static Point3D max(Point3D[] ps) { // metodo statico
19        Point3D max = zero; // ricerca max
20        for (Point3D elem : ps) {
21            if (elem.getSquaredModulus() > max.getSquaredModulus()) {
22                max = elem;
23            }
24        }
25        return max;
26    }
27 }
```

Uso Point3D

```
1 class UsePoint3D {
2     public static void main(String[] s) {
3         // creo e inizializzo vari punti
4         Point3D p1 = new Point3D();
5         p1.build(10, 20, 30);
6         Point3D p2 = new Point3D();
7         p2.build(5, 6, 7);
8         Point3D p3 = new Point3D();
9         p3.build(100, 100, 100);
10        Point3D p4 = Point3D.zero; // questo è lo zero
11
12        // costruisco l'array
13        Point3D[] array = new Point3D[]{p1, p2, p3, p4};
14
15        // calcolo il max
16        Point3D max = Point3D.max(array);
17
18        // stampo
19        System.out.println("Max: " + max.x + "," + max.y + "," + max.z);
20    }
21 }
```

Point3D: commenti

Si notino le due diverse chiamate

- `build` è sempre chiamata su un oggetto, ossia su: `new Point3D()`
- `zero` e `max` sono chiamate sulla classe `Point3D`

Razionale

- `max` è una funzionalità sui `Point3D`, quindi è naturale includerla nella classe `Point3D`, ma non è inquadrabile come funzionalità di un singolo oggetto (ognuno darebbe la stessa risposta)
- `zero` è una sorta di “costante”, quindi non è proprietà di un oggetto

Un errore comune

- se si omette l'indicazione del receiver, si rischia di chiamare una proprietà non statica da un metodo statico, e questo comporta un errore segnalato dal compilatore

⇒ a questo si ovvia inserendo sempre l'indicazione del receiver

Variante Point3DBis...inizializzazione con ritorno

```
1 class Point3DBis { // dichiarazione classe
2     double x; // 3 campi
3     double y;
4     double z;
5
6     Point3DBis build(double a, double b, double c) { // build con ritorno
7         this.x = a;
8         this.y = b;
9         this.z = c;
10        return this;
11    }
12
13    double getModuloQuadro() {
14        return this.x * this.x + this.y * this.y + this.z * this.z;
15    }
16
17    static Point3DBis zero = new Point3DBis().build(0, 0, 0);
18
19    static Point3DBis max(Point3DBis[] ps) { // metodo statico
20        Point3DBis max = zero; // ricerca max
21        for (Point3DBis elem : ps) {
22            if (elem.getModuloQuadro() > max.getModuloQuadro()) {
23                max = elem;
24            }
25        }
26        return max;
27    }
28 }
```

Use Point3DBis

```
1 class UsePoint3DBis {
2     public static void main(String[] s) {
3         // creo vari punti
4         Point3DBis p1 = new Point3DBis().build(10, 20, 30);
5         Point3DBis p2 = new Point3DBis().build(5, 6, 7);
6         Point3DBis p3 = new Point3DBis().build(100, 100, 100);
7         Point3DBis p4 = Point3DBis.zero; // questo è lo zero
8
9         // costruisco l'array
10        Point3DBis[] array = new Point3DBis[] { p1, p2, p3, p4 };
11
12        // calcolo il max
13        Point3DBis max = Point3DBis.max(array);
14
15        // stampo
16        System.out.println("Max: " + max.x + "," + max.y + "," + max.z);
17    }
18 }
```

Si noti la struttura del metodo build

- inizializza il valore dei campi
- restituisce `this`

Razionale

- è possibile concatenare creazione di oggetto e sua inizializzazione
- schema chiamato “fluente”

Sull'uso delle proprietà `static`

Consigli generali

- nella programmazione OO pura, andrebbero interamente evitati
- frequente invece l'uso di costanti
- a volte sono comodi, per ora cercare di tenere separate nelle classi le parti `static` dalle altre (poi vedremo “pattern” più sofisticati)
- le librerie di Java ne fanno largo uso (troppo largo forse)

Prassi frequente nelle librerie di Java

- se XYZ è una classe usata per generare oggetti, la classe XYZs conterrà solo proprietà statiche relative
- es: `Object/Objects`, `Array/Arrays`, `Collection/Collections`

In altri linguaggi... come Scala

- parte statica e non-statica **vanno necessariamente** posizionate in porzioni diverse del codice

Ristrutturazione Point3D

```
1 class Point3D {  
2     double x;  
3     double y;  
4     double z;  
5  
6     Point3D build(double a, double b, double c) {  
7         this.x = a;  
8         this.y = b;  
9         this.z = c;  
10        return this;  
11    }  
12  
13    double getSquaredModulus() {  
14        return this.x * this.x + this.y * this.y + this.z * this.z;  
15    }  
16 }
```

Modulo Points

```
1 class Points { // Modulo con funzionalità per punti
2
3     static Point3D zero = new Point3D().build(0, 0, 0);
4
5     static Point3D max(Point3D[] ps) { // metodo statico
6         Point3D max = zero; // ricerca max
7         for (Point3D elem : ps) {
8             if (elem.getSquaredModulus() > max.getSquaredModulus()) {
9                 max = elem;
10            }
11        }
12        return max;
13    }
14 }
```

Use Point3D e Points

```
1 class UsePoint3D {
2     public static void main(String[] s) {
3         // creo vari punti
4         Point3D p1 = new Point3D().build(10, 20, 30);
5         Point3D p2 = new Point3D().build(5, 6, 7);
6         Point3D p3 = new Point3D().build(100, 100, 100);
7         Point3D p4 = Points.zero; // questo è lo zero
8
9         // costruisco l'array
10        Point3D[] array = new Point3D[] { p1, p2, p3, p4 };
11
12        // calcolo il max
13        Point3D max = Points.max(array);
14
15        // stampo
16        System.out.println("Max: " + max.x + "," + max.y + "," + max.z);
17    }
18 }
```

Outline

- 1 Codice statico
- 2 Costruttori**
- 3 Controllo d'accesso
- 4 Distruzione oggetti
- 5 Una applicazione

La costruzione degli oggetti

L'operatore `new`

- Allo stato attuale delle nostre conoscenze, crea un oggetto inizializzando tutti i suoi campi al loro valore di default (p.e. 0 per i numeri), e ne restituisce il riferimento
- Altri metodi devono gestire la inizializzazione vera e propria
- Problema: garantire una corretta inizializzazione

Costruttori di una classe

- Assomigliano per struttura ai metodi
- Hanno lo stesso nome della classe in cui si trovano
- Nessun tipo di ritorno, possono avere dei parametri formali
⇒ alla `new` si possono quindi passare dei valori
- Il costruttore di default (a zero argomenti) è implicitamente definito solo se non se ne aggiungono altri – ecco perché era consentito scrivere: `new Point3D()`

Esempio: Point3D

```
1 class Point3D {    // dichiarazione classe
2     double x;
3     double y;
4     double z;
5     Point3D(double inx, double iny, double inz){ //costruttore
6         this.x = inx; // metto l'argomento inx in this.x
7         this.y = iny; // ..
8         this.z = inz; // ..
9     }
10    ...
11 }

12
13 //creo l'oggetto usando il costruttore a tre argomenti
14 Point3D p = new Point3D(10.0,20.0,30.0);
15 // stampo
16 System.out.println("p: " + p.x + "," + p.y + "," + p.z);
17 // costruttore di "default" in questo caso non funziona!
18 //Point3D p2=new Point3D(); NO!!
```

Gli argomenti del costr. spesso corrispondono ai campi

```
1 class Point3D {    // dichiarazione classe
2     double x;
3     double y;
4     double z;
5     Point3D(double x,double y,double z){
6         this.x = x; // metto l'argomento x in this.x
7         this.y = y; // ..
8         this.z = z; // ..
9     }
10    ...
11 }
12
13 //creo l'oggetto usando il costruttore a tre argomenti
14 Point3D p = new Point3D(10.0,20.0,30.0);
15 // stampo
16 System.out.println("p: " + p.x + "," + p.y + "," + p.z);
```

Esempio: Person (3 costruttori)

```
1 class Person { // dichiarazione classe
2     static int currentYear = new java.util.Date().getYear();
3     String name;
4     int birth;
5     boolean married;
6
7     Person(String name) {
8         this.name = name;
9         this.birth = Person.currentYear;
10        this.married = false;
11    }
12
13    Person(String name, int birth) {
14        this.name = name;
15        this.birth = birth;
16        this.married = false;
17    }
18
19    Person(String name, int birth, boolean married) {
20        this.name = name;
21        this.birth = birth;
22        this.married = married;
23    }
24 }
```

Esempio: Uso di Person (3 costruttori)

```
1 class UsePerson {  
2     public static void main(String[] s) {  
3         // Persona p1=new Persona(); NO!!  
4         Person p2 = new Person("Mario Rossi");  
5         Person p3 = new Person("Gino Bianchi", 1979);  
6         Person p4 = new Person("Carlo Verdi", 1971, true);  
7     }  
8 }
```

Sequenza d'azioni effettuate con una new

- si crea l'oggetto con tutti i campi inizializzati al default
- si esegue il codice del costruttore (`this` punta all'oggetto creato)
- la `new` restituisce il riferimento `this`

Istruzione `this(...)`

Usabile per chiamare un altro costruttore

- tale istruzione può solo essere la prima di un costruttore
- questo meccanismo consente di riusare il codice di altri costruttori

```
1 class Person2 { // dichiarazione classe
2     static int currentYear = new java.util.Date().getYear();
3     String name;
4     int birth;
5     boolean married;
6
7     Person2(String name, int birth, boolean married) {
8         this.name = name;
9         this.birth = birth;
10        this.married = married;
11    }
12
13    Person2(String name, int birth) { // richiama costruttore a 3 arg..
14        this(name, birth, false);
15    }
16
17    Person2(String name) {
18        this(name, Person2.currentYear); // richiama costruttore a 2 arg..
19    }
20 }
```

Overloading dei costruttori

Overloading

- È la tecnica che consente di dare più significati ad uno stesso “nome”
- L'overloading è un meccanismo importante per il programmatore
- La difficoltà che comporta è dovuta alla tecnica di disambiguazione
- Due esempi visti finora: overloading operatori matematici, overloading costruttori

Overloading dei costruttori

Data una `new`, quale costruttore richiamerà? (JLS 15.12)

- si scartano i costruttori con numero di argomenti che non corrisponde
- si scartano i costruttori il cui tipo d'argomenti non è compatibile
- se ve ne è uno allora è quello che verrà chiamato...
- altrimenti il compilatore segnala un errore

Overloading dei metodi

Overloading dei metodi

- Si può fare overloading dei metodi con stessa tecnica di risoluzione
- Sia su metodi statici, che su metodi standard (metodi istanza)

```
1 class ExampleOverloading {
2     static int m(double a, int b) {
3         return 1;
4     }
5
6     static int m(int a, double b) {
7         return 2;
8     }
9
10    static int m2(double a, double b) {
11        return 1;
12    }
13
14    static int m2(int a, int b) {
15        return 2;
16    }
17
18    public static void main(String[] s) {
19        // System.out.println(""+m(1,1)); AMBIGUOUS!
20        // System.out.println(""+m(1.5,1.5)); NO COMPATIBLE!
21
22        System.out.println("" + m2(1.5, 1.5)); // 1
23        System.out.println("" + m2(1, 1)); // 2
24    }
```


Outline

- 1 Codice statico
- 2 Costruttori
- 3 Controllo d'accesso**
- 4 Distruzione oggetti
- 5 Una applicazione

I package

Problema

- Un framework mainstream come quello di Java può disporre di decine di migliaia di classi di libreria e di applicazione
- È necessario un meccanismo per consentire di strutturarle in gruppi (a più livelli gerarchici)
- Per poter meglio gestirli in memoria secondaria, e per meglio accedervi

Package in Java

- Ogni classe appartiene ad un **package**
- Un package ha nome gerarchico $n1.n2 \dots nj$ (p.e. `java.lang`)
- Le classi di un package devono trovarsi in una medesima directory
- Questa è la subdirectory $n1/n2/ \dots /nj$ a partire da una delle directory presenti nella variabile d'ambiente `CLASSPATH`

I package pt. 2

Dichiarazione del package

- Ogni unità di compilazione (file .java) deve specificare il suo package
- Lo fa all'inizio del sorgente con la direttiva `package` pname;
- Se non lo fa, allora trattasi del package di "default"
- Se lo fa, tale file dovrà stare nella opportuna directory

Importazione classi da altri package

- Una classe va usata (p.e. nella `new` o come tipo) specificando anche il package (p.e. `new java.util.Date()`)
- Per evitare tale specifica, si inserisce una direttiva di importazione
 - ▶ `import java.util.*;` importa tutte le classi di `java.util`
 - ▶ `import java.util.Date;` importa la sola `java.util.Date`

Unità di compilazione e livello d'accesso "package"

Unità di compilazione

- È un file compilabile in modo atomico da javac
- Si deve chiamare con estensione .java
- Può contenere varie classi indicate una dopo l'altra

Livello d'accesso package

- Le classi in una unità di compilazione, i loro metodi, campi, e costruttori hanno di default il **livello d'accesso package**
- ⇒ sono visibili e richiamabili solo dentro al package stesso
- ⇒ sono invisibili da fuori

Livelli d'accesso `public` e `private`

Livello d'accesso `public` – visibile da tutte le classi

- Lo si indica anteponendo alla classe/metodo/campo/costruttore la keyword `public`
- La corrispondente classe/metodo/campo/costruttore sarà visibile ed utilizzabile da qualunque classe, senza limitazioni

Livello d'accesso `private` – visibile solo nella classe corrente

- Lo si indica anteponendo al metodo/campo/costruttore la keyword `private`
- Il corrispondente metodo/campo/costruttore sarà visibile ed utilizzabili solo dentro alla classe in cui è definito

Qualche conseguenza

A livello di classe

- In una unità di compilazione solo una classe può essere **public**, e questa deve avere lo stesso nome del file `.java`

A livello di metodi/campi/costruttori

- la scelta **public/private** può consentire di gestire a piacimento il concetto di **information hiding**, come approfondiremo la prossima settimana

La keyword `final`

`final` = non modificabile

- Ha vari utilizzi: in metodi, campi, argomenti di funzione e variabili
- Tralasciamo per ora il caso dei metodi
- Negli altri casi denota variabili/campi assegnati e non più modificabili

Il caso più usato: costanti di una classe

- Es.: `public static final int CONST=10;`
- Perché usarle?
 - ▶ Anche se `public`, si ha la garanzia che nessuno le modifichi
 - ▶ Sono gestibili in modo più performante
 - ▶ Nota la formattazione

Il caso dei “magic number”

- Ogni numero usato in una classe per qualche motivo (3,21,...)...
- ...è sarebbe opportuno sia mascherato da una costante, per motivi di leggibilità e di più semplice modificabilità

Esempio Magic Numbers

```
1 public class MagicExample {  
2     // Put 100 into a constant and give it a name!!  
3     private static final int SIZE = 100;  
4  
5     public static void main(String[] s) {  
6         double[] array = new double[SIZE];  
7         double sum = 0;  
8         for (int i = 0; i < SIZE; i++) {  
9             // Assegno un numero random  
10            array[i] = Math.random();  
11            sum = sum + array[i];  
12        }  
13        System.out.println("Somma " + sum);  
14    }  
15 }
```


GuessMyNumberApp revisited

```
1 import java.io.Console; // Classe non funzionante dall'IDE Eclipse
2 import java.util.Random;
3
4 public class GuessMyNumberApp {
5
6     public static final int ATTEMPTS = 10;
7     public static final int MAX_GUESS = 100;
8     public static final int MIN_GUESS = 1;
9
10    public static void main(String[] args) {
11        int number = new Random().nextInt(MAX_GUESS - MIN_GUESS) + MIN_GUESS;
12        for (int i = 1; i <= ATTEMPTS; i++){
13            System.out.println("Attempt no. "+i);
14            System.out.println("Insert your guess.. ");
15            int guess = Integer.parseInt(System.console().readLine());
16            if (guess == number){
17                System.out.println("You won!!");
18                return;
19            } else if (guess > number){
20                System.out.println("Your guess is greater..");
21            } else {
22                System.out.println("Your guess is lower..");
23            }
24        }
25        System.out.println("Sorry, you lost!");
26    }
27 }
```

Outline

- 1 Codice statico
- 2 Costruttori
- 3 Controllo d'accesso
- 4 Distruzione oggetti**
- 5 Una applicazione

Allocazione degli oggetti in memoria

Operatore `new`

- Incorpora l'equivalente della funzione `malloc` del C
- Il compilatore calcola la dimensione necessaria da allocare
- La `new` chiama il gestore della memoria, che alloca lo spazio necessario
- Si inizializza l'area e si restituisce il suo riferimento

Cosa c'è in un oggetto (e, quanto è grande?)

- Non tutti i dettagli sono noti, dipende dalla JVM
- Di sicuro contiene i seguenti elementi:
 - ▶ Spazio per ogni campo (non statico) della classe
 - ▶ Un riferimento ad una struttura dati relativa alla classe dell'oggetto
 - ▶ Un riferimento alla **tabella dei metodi virtuali** (una struttura dati necessaria a trovare dinamicamente i metodi da richiamare)

Distruzione degli oggetti

Il tempo di vita degli oggetti

- Durante l'esecuzione di un programma, è verosimile che molto oggetti vengano creati
 - Ogni creazione comporta l'uso di una parte di memoria centrale
 - Non è noto quanto durerà l'esecuzione del programma
- ⇒ Qualcuno dovrà preoccuparsi di deallocare la memoria

Il garbage collector (GC)

- E' un componente della JVM richiamato dalla JVM con una frequenza che dipende dallo stato della memoria
- Ogni volta, cerca oggetti in memoria heap che nessuna parte attiva del programma (thread) sta più usando (neanche indirettamente)
- Trovatili, li dealloca (come la free del C) **senza che il programmatore debba occuparsene**

Esempio funzionamento GC

```
1 class GC {
2     private static long size = 1000;
3
4     public static void main(String[] s) {
5         // Runtime dà info sull'esecuzione
6         Runtime r = Runtime.getRuntime();
7
8         // Creo oggetti all'infinito
9         for (long l = 0; true; l++) {
10             new Object();
11             // Stampo solo ogni tanto
12             if (l % size == 0) {
13                 System.out.print("Objs (*10^6): " + l / 1000000);
14                 System.out.println(" Freemem (MB):" + (r.freeMemory() >> 20));
15             }
16             // La memoria libera si vedrà calare lentamente
17             // e poi riprendersi di colpo, ciclicamente
18         }
19     }
20 }
```

Outline

- 1 Codice statico
- 2 Costruttori
- 3 Controllo d'accesso
- 4 Distruzione oggetti
- 5 Una applicazione**

Applicazione: Mandelbrot

Problema

Data una semplice classe `Picture` che gestisce gli aspetti grafici, realizzare una applicazione che disegna il frattale Mandelbrot.

Elementi progettuali

- Classe fornita `Picture` – codice non comprensibile ora
 - ▶ ha un costruttore che accetta larghezza e altezza in pixel della finestra
 - ▶ metodo `void drawPixel(int x, int y, int color)`
- Classe `Complex` modella numeri complessi e operazioni base
- Classe `Mandelbrot` si occupa di calcolare il valore di ogni punto del rettangolo
 - ▶ metodo `void advancePosition()` passa al prossimo punto
 - ▶ metodo `boolean isCompleted()` dice se ci sono altri punti da calcolare
 - ▶ metodo `int computeIterations()` dice quante iterazioni vengono calcolate per il punto corrente
- Classe `MandelbrotApp` ha il solo `main`

Elementi implementativi

- Implementazione ancora preliminare e da migliorare

Classe Complex

```
1 public class Complex {
2
3     double re;
4     double im;
5
6     Complex(double re, double im) {
7         this.re = re;
8         this.im = im;
9     }
10
11     double getScalarProduct(){
12         return this.re * this.re + this.im * this.im;
13     }
14
15     // Crea un nuovo complesso, sommando this a c
16     Complex sum(Complex c){
17         return new Complex(this.re + c.re, this.im + c.im);
18     }
19
20     // Crea un nuovo complesso, moltiplicando this a c
21     Complex times(Complex c){
22         return new Complex(this.re * c.re - this.im * c.im,
23                             c.re * this.im + c.im * this.re);
24     }
25
26 }
```


Classe Mandelbrot

```
1 public class Mandelbrot {
2
3     static final double MAX_PRODUCT = 4.0;
4     int width, height, x, y; // uno per linea di norma
5     double minx, maxx, miny, maxy, maxIter; // uno per linea di norma
6
7     Mandelbrot(int width, int height, double minx, double maxx,
8               double miny, double maxy, int maxIter) {
9         this.width = width; this.height = height;
10        this.minx = minx; this.maxx = maxx;
11        this.miny = miny; this.maxy = maxy;
12        this.maxIter = maxIter;
13    }
14    void advancePosition(){
15        x = (x + 1) % width;
16        y = y + (x == 0 ? 1 : 0);
17    }
18    boolean isCompleted(){
19        return y == height;
20    }
21    int computeIterations(){
22        Complex c0 = new Complex(this.minx + (this.maxx - this.minx) * x / width,
23                                this.miny + (this.maxy - this.miny) * y / height);
24
25        Complex c = c0;
26        int iter;
27        for (iter = 0; c.getScalarProduct() < MAX_PRODUCT && iter < this.maxIter; iter++) {
28            c = c.times(c).sum(c0); // c = c*c + c0
29        }
30        return iter;
31    }
```

Classe MandelbrotApp

```
1 public class MandelbrotApp {
2
3     public static final int WIDTH = 800;
4     public static final int HEIGHT = 800;
5     public static final double MINX = -1.5;
6     public static final double MAXX = 0.5;
7     public static final double MINY = -1.0;
8     public static final double MAXY = 1.0;
9     public static final int MAX_ITER = 32;
10
11     public static int greyColorFromIterations(int iter, int maxIter){
12         if (iter == MAX_ITER) { // Out of Mandelbrot set
13             return 0;
14         }
15         iter = 255-iter*(256/MAX_ITER); // 255,254,...,0 colors
16         return iter | iter << 8 | iter << 16; // as grey
17     }
18
19     public static void main(String[] s){
20
21         Mandelbrot mb = new Mandelbrot(WIDTH,HEIGHT,MINX,MAXX,MINY,MAXY,MAX_ITER);
22         Picture p = new Picture(WIDTH,HEIGHT);
23         while (!mb.isCompleted()){
24             int iter = mb.computeIterations();
25             int color = greyColorFromIterations(iter,MAX_ITER);
26             p.drawPixel(mb.x, mb.y, color);
27             mb.advancePosition();
28         }
29     }
30 }
31
32 }
```

Preview del prossimo laboratorio

Obbiettivi

- Esercizi su piccoli algoritmi su array e tipi primitivi
- Uso costruttori
- Aspetti avanzati della compilazione in Java