2018
Programmazione a oggetti
parte 1

Giorgio Bruno

Dip. Automatica e Informatica
Politecnico di Torino
email: giorgio.bruno@polito.it

Presentazione

Il corso presenta le tecniche di sviluppo del software basate sul paradigma a oggetti mediante lo studio del linguaggio java (versione 8).

Le abilità che saranno acquisite riguardano:

- la padronanza di java,
- la capacità di analizzare i requisiti di un sistema software di complessità ragionevole allo scopo di definire un idoneo progetto a oggetti,
- la capacità di implementare e collaudare le classi del progetto mediante l'ambiente di sviluppo *eclipse*.

In aggiunta alle lezioni ed esercitazioni in aula si svolgeranno delle esercitazioni in laboratorio che riguardano la soluzione di alcuni semplici casi di studio mediante la scrittura di programmi java con l'ambiente eclipse.

L'esame si svolge in laboratorio e consiste nella scrittura di un programma java che realizzi i requisiti forniti.

Evoluzione dei linguaggi

Moduli: raggruppamento di dati e operazioni con vari livelli di visibilità; principio di information hiding.

Classi e oggetti: le caratteristiche degli individui (oggetti) sono definite nelle classi di appartenenza.

Linguaggi a oggetti propriamente detti (*object-oriented*): le classi sono imparentate mediante *inheritance* (ereditarietà) singola o multipla.

Linguaggi a oggetti fortemente tipizzati (strongly typed): tutti i dati hanno un tipo che stabilisce le regole di uso. Esempi: Java, C++, C#.

Programmazione a oggetti

Un oggetto è più di un dato ed è più di un'operazione; è l'integrazione dei dati e delle operazioni rivolte a quei dati.

Ad esempio, un ordine d'acquisto può essere rappresentato da un oggetto che contiene varie informazioni (sui prodotti, l'ordinante, le modalità di consegna) e le operazioni di gestione dell'ordine. Quindi è un *building block* che può essere usato in varie applicazioni.

Dati e operazioni (metodi) hanno vari livelli di visibilità e ciò conferisce robustezza al programma.

A run time il programma è una rete di oggetti collegati tra loro mediante associazioni. Gli oggetti interagiscono: un metodo eseguito da A può chiamare un metodo di B.

Principi generali

Gli **oggetti** sono entità dinamiche, generate e distrutte nel corso dell'esecuzione del programma.

Gli oggetti sono definiti mediante **classi**. Possiedono dati (**attributi**) ed eseguono operazioni (**metodi**).

Un programma consiste di componenti (*classi*, *interfacce*, *tipi enumerativi*) raggruppati in **package**.

L'esecuzione è basata sulla chiamata di funzioni dette metodi; i blocchi di attivazione dei metodi sono tenuti nello *stack* mentre gli oggetti si trovano in una struttura detta *heap*.

Paradigma a oggetti

Il paradigma a oggetti comprende anche il progetto a oggetti e l'analisi a oggetti (object-oriented design e object-oriented analysis).

Nell'analisi a oggetti l'obiettivo è la definizione di una struttura di classi idonea a formalizzare i requisiti del sistema da sviluppare.

Le classi sono viste in un'ottica *business-oriented* e devono rappresentare i concetti degli utenti del sistema.

Nel progetto a oggetti si definisce un'architettura di classi che sarà l'input dell'implementazione. In questa fase si utilizzano pattern (schemi consolidati).

L'UML fornisce una notazione standard per rappresentare strutture con classi e relazioni.

Java

Java è un linguaggio robusto, versatile, in continua evoluzione.

Inoltre è portabile: un programma java è tradotto in un formato intermedio unico che poi può essere eseguito da vari interpreti a seconda dell'ambiente target.

Ha una ricca libreria di classi predefinite e lo sviluppo dei programmi è supportato da strumenti efficaci come *eclipse*.

Organizzazione del corso

Lezioni in aula (Bruno e Basile)

Esercitazioni in aula (Basile)

Esercitazioni in laboratorio (Regano) + borsisti

5 Esercitazioni di laboratorio per 3 squadre con il calendario e la suddivisione in squadre indicati nella pagina del corso.

Argomenti delle 5 Es.

array

inheritance

collezioni

stream

file

Lezioni Bruno

Parte 1

Aspetti di base
Classi e oggetti
Package
Classi predefinite: String …
Array
Ereditarietà
Classi astratte
Eccezioni
Classe Object
Classi Math e System
Tipi enumerativi

Parte 2

Interfacce
Tipi generici
Ordinamenti
Interfacce funzionali
Espressioni lambda
Interfaccia Comparator <T>
Collezioni: set, liste, mappe
Stream
Classe Collectors

Lezioni Bruno

Parte 3

Classe Object

Formattazione

Classe Scanner

Gestione file

Espressioni regolari

Date

Thread

Parte 4

Cenni di Ingegneria del software

Software life cycle

Sviluppo agile

Sviluppo evolutivo del processo software; CMM

Qualità del software

Misure

Modelli UML: class models

Pattern

Parte 5

Esempi finali: analisi e progetto a oggetti

Lezioni Basile

Ambiente Java, Eclipse

Gestione configurazione; Subversion

Testing; JUnit

Interfacce grafiche; swing

Note sull'esame

L'esame consiste in una prova al calcolatore (nei laib) della durata di 2 ore.

La prova è suddivisa in due parti:

Sviluppo di un programma in Java, utilizzando l'ambiente Eclipse (peso sul voto finale ~85%).

Domande di verifica teorica su argomenti (Ingegneria del software e caratteristiche generali di Java) trattati a lezione (peso sul voto finale ~15%).

Non è possibile utilizzare copie cartacee di slide, libri, telefoni cellulari, penne USB, etc.

Note sull'esame

Per poter sostenere l'esame occorre essere in grado di:

effettuare il login su PC dei LAIB con le credenziali ufficiali polito,

utilizzare l'ambiente Eclipse,

effettuare check-out e commit di un progetto contenuto in un repository SVN,

scrivere programmi in Java.

Durante l'esame non sarà fornita assistenza su questi aspetti.

Note sull'esame

La valutazione si basa su due diverse versioni del programma:

la versione lab; quella presente nel repository alla scadenza della prova di laboratorio (conta l'ultimo commit);

la version home: quella presente nel repository alla scadenza della consegna da casa.

Il voto dipende

- dalla percentuale di test superati dalla versione lab;
- dal numero di modifiche della versione home rispetto alla versione lab.

Se il programma home non supera tutti i test, non sarà valutato e lo studente verrà considerato ritirato.

Java

Aspetti di base

Tipi predefiniti e operatori

Istruzioni di controllo del flusso di esecuzione

Valori e oggetti

Java tratta valori e oggetti.

I valori appartengono ai tipi predefiniti come int, char, boolean.

Gli oggetti sono istanze di classi predefinite o definibili dall'utente.

Le classi predefinite si trovano in una libreria organizzata in packages, tra i quali

java.lang (libreria standard)

java.util

java.io

Tipi predefiniti

Valori numerici:

interi: **byte**, **short** (16 bit), **int** (32), **long** (64).

decimali: **float**, **double**

Caratteri:

char, rappresentazione Unicode 16 bit

Booleani: **boolean**

Costanti

int:123 long: 1234L float: 56.7F double 78.9 1.534e3

char: 'c'

boolean: true, false

Operatori

Aritmetici: +, -, *, /, %. Pre (post) incremento/decremento: ++, --.

Confronto: ==, !=, >, >=, <, <=.

Gli operatori == e != confrontano valori oppure indirizzi di memoria nel caso di oggetti. Per confrontare i contenuti di due oggetti si usa il metodo *equals*.

Operatori booleani: and e or con valutazione accorciata (&&, ||) oppure completa (&, |), not (!) xor (^) [vale true se soltanto uno dei due operandi è true].

Operatori su bit.

Operatore *ternario*: *e*? *e1*:*e2*. Se l'espressione *e* è vera, il risultato è l'espressione *e1* altrimenti è l'espressione *e2*.

Concatenazione di stringhe: +. Esempio: "hello " + "world!".

Assegnazioni: =, +=, -=,*=, /=,%=.

Conversioni

Conversione di valori numerici mediante casting:

int i = 10; double d = 15.1;

float f; // inizializzato a 0

f = i + d; // dà errore

f = (float) (i + d);

Istruzioni di controllo di sequenza

if-then, if-then-else, switch

for, while, do-while

break, continue, return

Esempio di metodo

```
static double calcolaRadiceQuadrata (double r) {
if (r < 0) r = -r;
double x = r/2;
double y = r/x;
double radice = (x+y)/2;
double delta = r-radice*radice;
if (delta < 0) delta = - delta;
while (delta > 0.01) {
        x = radice; y = r/x;
        radice = (x+y)/2;
        delta = r-radice*radice;
        if (delta < 0) delta = - delta;
}
return radice;
}
```

soltanto valori in questo esempio

Sviluppo di programmi java

Kit di sviluppo: Java Standard Edition (Java SE)

contiene compilatore (*javac*), interprete (*java*) e *libreria* delle classi predefinite (organizzata in package).

download di java SE (Standard Edition) ultima versione Java SE 9.0.4

http://www.oracle.com/technetwork/java/javase/downloads/index.html

Ambiente integrato **eclipse**

download di eclipse ultima versione **Oxygen**

https://www.eclipse.org/downloads/

Classi

Struttura

Esempi

Proprietà

Esempio di classe

La classe **Point** rappresenta punti interi nel piano.

Requisiti

La classe fornisce un *costruttore* che permette di generare un oggetto di tipo Point date le coordinate x e y,

e dei *metodi* per leggere ciascuna coordinata,

per spostare un punto (date le nuove coordinate)

e per ottenere una *rappresentazione testuale* del punto, ad es. "x = 10 y = 20" per un punto di coordinate 10 e 20.

Nota: la classe Point sta nel *package* esempiPrimaParte.

Un package è una collezione di classi.

La prima istruzione di una classe indica il package di appartenenza:

```
package esempiPrimaParte;
```

Class Point

```
package esempiPrimaParte;
public class Point {
   private int x,y; // attributi d'istanza inizializzati a 0
         public int getX() {return x;} // metodi getter
         public int getY() {return y;}
   public Point (int x1, int y1) {x = x1; y = y1;} // costruttore
   public void move (int x1, int y1) {x = x1; y = y1;} // metodo d'istanza
   public String toString (){return "x = " + x + " y = " + y;}
}
```

Class Point

Da notare:

- il costrutto **class**
- i livelli di visibilità: **public** (visibile da tutte le classi)
  private (soltanto nella classe corrente)
- il costruttore: nome della classe seguito da parametri tra ()
- il metodo toString (ereditato da Object e ridefinito)

main di prova

```
package esempiPrimaParte;
public class Test {
public static void main(String[] args) {
        Point p1 = new Point(23, 94);  // istanziazione
        // genera il punto (23,94) chiamando il costruttore
        System.out.println(p1.getX());
                // visualizza su schermo la coordinata x
        p1.move(10,20);     // muove il punto, cambiandone le coordinate
        System.out.println(p1);
        // chiama implicitamente toString e quindi stampa: x = 10 y = 20
}
```

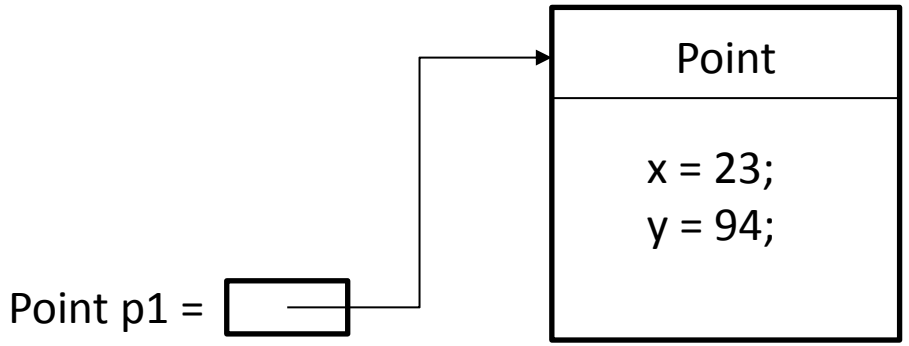
main di prova

Da notare:

- il metodo statico main, che si può mettere in Point o in *un'altra classe*
- l'operatore **new**
- la *variabile* p1 che può contenere un riferimento ad un oggetto Point
- la classe **System**
- l'oggetto statico (denotato da) System.out che rappresenta lo standard output
- il metodo println
- la *notazione puntata* per chiamare il metodo di un oggetto, es. p1.getX()

Stack e heap

stack: blocchi di attivazione dei metodi
heap: oggetti



Point p1 = new Point(23, 94);

p1 è una variabile che contiene l'indirizzo dell'oggetto Point generato dal costruttore

Passi:
1. si genera un punto con gli attributi x e y a 0;
2. si copiano i parametri attuali negli attributi.

public **Point** (int x1, int y1) {x = x1; y = y1;}

Class Point (variante)

```
package esempiPrimaParte;
public class Point {
    private int x,y; // attributi d'istanza
    public int getX() {return x;} // metodi getter
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public void move(int x, int y) {this.x = x; this.y = y;}
    public String toString(){return "x = " + x + " y = " + y;}
            // metodo d'istanza ereditato da Object e ridefinito
}
```

I parametri hanno gli stessi nomi degli attributi degli oggetti Point. Per distinguere i secondi dai primi si usa la keyword **this** che denota l'oggetto corrente.

*Note su **this***

La keyword this denota l'oggetto corrente ed è usata nei casi seguenti:

- quando un attributo e un parametro formale hanno nomi identici;
- quando si vuole passare l'oggetto corrente come parametro o come risultato di un metodo;
- quando un costruttore vuole servirsi di un altro costruttore della stessa classe.

Passaggio dell'oggetto corrente

Aggiungere i metodi send e receive usati come segue.

```
Point p1 = new Point(10, 20);
Point p2 = new Point(100, 200);
p1.send(p2);
```

Il metodo send manda il punto corrente (p1 nell'es.) al punto passato come parametro (p2). Il punto ricevente stampa il punto ricevuto.

Il metodo send chiama il metodo receive del ricevente.

Classe Point

```
public class Point {
    private int x,y;
    public int getX() {return x;}
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public void move(int x, int y) {this.x = x; this.y = y;}
    public String toString(){return "x = " + x + " y = " + y;}

    public void send(Point p) {p.receive(this);}
    public void receive(Point p) {System.out.println(p);}

}
```

Chiamata di un altro costruttore

Dato il costruttore già definito

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

si aggiunga un costruttore per generare un punto sulla diagonale y = x.

Si può scrivere in due modi

```
public Point(int a) {x = a; y = a;}
```

oppure

```
public Point(int a) {this(a,a);}
```

Nel secondo caso il costruttore chiama un costruttore già esistente e il match si basa sulla sequenza dei tipi dei parametri (nell'es. la sequenza è int int).

La chiamata del costruttore deve essere la prima istruzione.

Class Rectangle

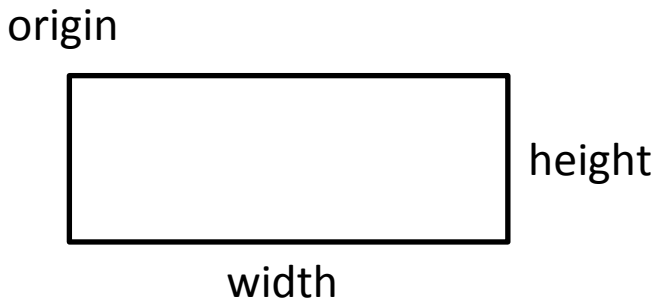
Requisiti

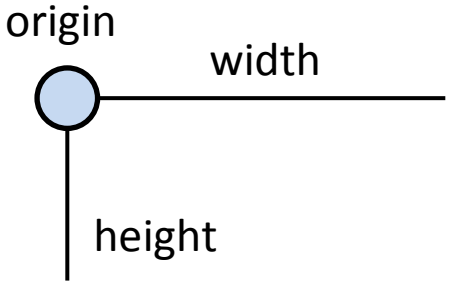
La classe Rectangle rappresenta rettangoli con i lati paralleli agli assi.

Un rettangolo è caratterizzato da larghezza e altezza (valori interi inizializzati a 20 e 10) e dal vertice in alto a sinistra (un oggetto Point).

Un rettangolo si costruisce dati il vertice e i lati oppure in modo default con il vertice nell'origine degli assi.

Un rettangolo si può spostare mediante il metodo move; inoltre, dato un rettangolo, si può ottenere l'area e una rappresentazione testuale (ad es. x = 15 y = 25 w = 100 h = 200).





Class Rectangle

```
package esempiPrimaParte;
public class Rectangle {
    private int width = 20; private int height = 10;
    private Point origin = null;
    public Rectangle () {origin = new Point(0, 0);}
    public Rectangle (Point p, int w, int h)
                    {origin = p; width = w; height = h;}

    public String toString(){
          return origin.toString() + " w = " + width + " h = " + height;}

    public void move (int x, int y) {origin.move (x,y);}
    public int area () {return width * height;}
}
```

chiama il metodo move del punto origin

Class Rectangle

Da notare:

- 2 costruttori
- l'attributo origin che può contenere un riferimento ad un oggetto
- un metodo che chiama un metodo di un altro oggetto.

Costruttore di default

Se in una classe non è definito alcun costruttore, il compilatore ne aggiunge uno privo di parametri e vuoto (di default).

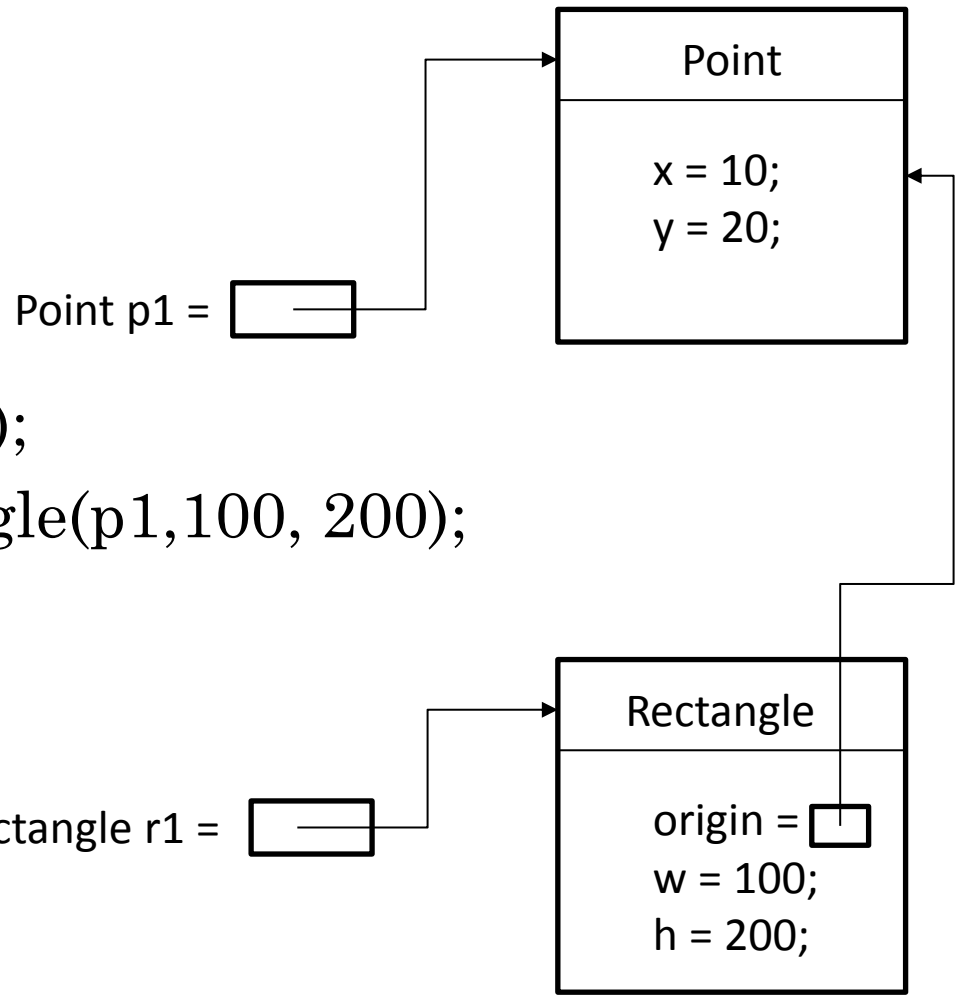
Es.

```
public Rectangle () {}
```

main di prova

```
public static void main(String[] args) {
   Point p1 = new Point(10, 20);
   Rectangle r1 = new Rectangle(p1, 100, 200);
   r1.move(15,25);
   System.out.println(r1); // x = 15 y = 25 w = 100 h = 200
   System.out.println(r1.area()); // 20000
}
```

Stack e heap



Point p1 = new Point(10, 20);

Rectangle r1 = new Rectangle(p1,100, 200);

p1 e r1 sono due variabili riferimento; x, y, origin, w, h sono attributi (di oggetti)

Il programma di test in un package separato (import)

```
package esempiPrimaParte;
public class Point {
 ……
}
```

La keyword **import** consente di importare le classi che servono, oppure tutte quelle del package indicato (con la wildcard *).

```
package testEsempiPrimaParte;
import esempiPrimaParte.*;

public class Test{

public static void main(String[] args) {
……
}

}
```

Note su Rectangle

Il costruttore public **Rectangle** (Point p, int w, int h)

{origin = p; width = w; height = h;}

riceve il punto origine dall'esterno e ne copia il riferimento nell'attributo origin. Il rettangolo non è l'unico proprietario del punto, quindi il punto può essere modificato dall'esterno a sua insaputa.

Per evitare ciò si può modificare il costruttore come segue (facendo una copia interna del punto origine)

```
public Rectangle (Point p, int w, int h)
{origin = new Point(p.getX(), p.getX()); width = w; height = h;}
```

oppure si può usare questo costruttore

```
public Rectangle (int x, int y, int w, int h)
{origin = new Point(x, y); width = w; height = h;}
```

Proprietà

Attributi e metodi sono proprietà.

Le proprietà possono essere d'istanza o di classe (anche dette **statiche**).

Una **proprietà d'istanza** si riferisce ad un oggetto mediante l'operatore ".".

Una **proprietà di classe** non si riferisce ad alcun oggetto; si indica facendone precedere il nome dalla classe di appartenenza seguita dal "."

Se usata nella stessa classe basta il nome.

Esempi:

1. definire un metodo che generi un nuovo punto come somma di due punti dati.
2. definire il punto statico origin (0,0).

Metodo statico (somma di due punti)

```
public class Point {
    private int x,y;
    public int getX() {return x;}
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public void move(int x, int y) {this.x = x; this.y = y;}
    public String toString(){return "x = " + x + " y = " + y;}
    public static Point sum (Point p1, Point p2) {
            return new Point (p1.x+p2.x,p1.y+p2.y);
    }
}
```

```
//main di test nella classe Point
public static void main(String[] args) {
        Point p1 = new Point(10, 20);
        Point p2 = new Point(100, 200);
        Point p3 = sum(p1,p2);  // stessa classe: basta sum
        System.out.println(p3); // x = 110 y = 220
}
```

Attributo statico

```
public class Point {
   private int x,y;
   public int getX() {return x;}
   public int getY() {return y;}
   public Point(int x, int y) {this.x = x; this.y = y;}
   public void move(int x, int y) {this.x = x; this.y = y;}
   public String toString(){return "x = " + x + " y = " + y;}
   public static Point sum (Point p1, Point p2) {
          return new Point (p1.x+p2.x,p1.y+p2.y);
   }
   public static Point origin = new Point(0,0);
}
```

```
public class Test {
        public static void main(String[] args) {
                Point p1 = new Point(10, 20);
                Point p3 = Point.sum(p1, Point.origin);
                //classe diversa: occorre premettere il nome della classe
        }}
```

Classi di libreria (in java.lang)

Stringhe

Classi Wrapper

Stringhe

classi String e StringBuilder

Un oggetto String contiene una sequenza non modificabile di caratteri.

Invece un oggetto StringBuilder contiene una sequenza modificabile di caratteri.

Come si genera una stringa:

String s1 = "hello ";

String s2 = s1 + "world!";

L'operatore + effettua la concatenazione.

Alcuni metodi di String

public char charAt(int index)

boolean **equals** (Object anObject)

// ereditata da Object, confronta il contenuto

public boolean equalsIgnoreCase(String anotherString)

public int **compareTo**(String anotherString)

// dà 0 se uguali, -1 (1) se anotherString segue (precede)

public boolean startsWith(String prefix, int toffset)

public int indexOf(int ch)

public String substring(int beginIndex, int endIndex)

public int **length**()

public String **toLowerCase**()

public String trim() //toglie gli spazi eventuali all'inizio e alla fine

public static String **valueOf**(int i)

// produce la stringa corrispondente

Confronti tra stringhe

```
System.out.println("alfa".equals("beta"));       //false

System.out.println("alfa".compareTo("beta"));          //-1
```

Alcuni metodi di StringBuilder

StringBuilder()
// genera una stringa vuota con capacità iniziale di 16 caratteri
StringBuilder (int capacity)
StringBuilder (String str)

Metodi
StringBuilder append (*vari tipi*)
insert

Danno come risultato l'oggetto corrente, cioè quello a cui si applica il metodo.

Ciò rende possibile una catena di chiamate, ad es.
new StringBuilder().append("alfa").append('+').append("beta");

Esempio

Scrivere un metodo statico per rovesciare una stringa.

```
public class C {
public static String invertiStringa(String s) {
        StringBuilder r = new StringBuilder();
        for (int i = s.length() - 1; i >= 0; i--) r.append(s.charAt(i));
        return r.toString();
}}

programma di prova in un'altra classe
public static void main(String[] args) {
        System.out.println(C.invertiStringa("hello world")); // dlrow olleh
}

oppure più semplicemente
StringBuilder sb = new StringBuilder("hello world").reverse(); //predefinito
```

Wrapper classes

Incapsulano valori di tipo predefinito:

numerici	Byte, Short, Integer, Long
	Float, Double
caratteri	Character
booleani	Boolean

Effettuano conversioni stringhe → valori di tipo predefinito

Mediante auto boxing/unboxing

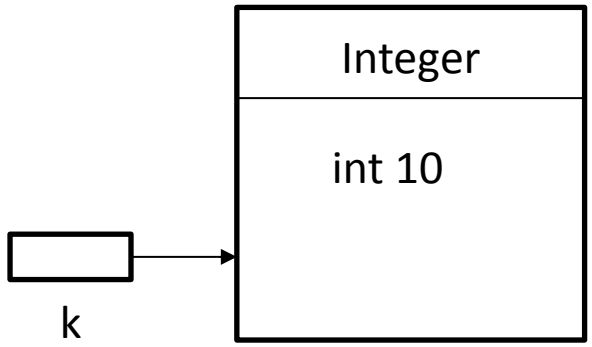
la conversione valori ↔ oggetti numerici è automatica

Gli oggetti di queste classi possono essere inseriti in collezioni.

Classi numeriche e valori

Esempi con Integer

Integer k = new Integer (10); // costruttore
oppure con auto boxing
Integer k = 10; int j = 100; k = j;



int i = k.intValue(); //intValue dà l'int contenuto nell'Integer
oppure con auto unboxing
int i = k;

Conversioni da stringhe

da String a Integer (2 modi: costruttore e valueOf)

String s = "123";

Integer h1 = new **Integer**(s); // si usa questo costruttore

oppure Integer h1 = Integer.**valueOf**(s); // metodo statico

Se s non è convertibile (ad es. s = "123a"), i metodi sollevano l'eccezione NumberFormatException.

da String a int (1 modo: parseInt) parseInt è un metodo statico

int int1 = Integer.**parseInt**(s); può sollevare lo stesso tipo di eccezione.

String -> double

String sd = "123.45";

Double dD = new **Double**(sd);

dD = Double.**valueOf**(sd);

double d = Double.**parseDouble**(sd);

Conversioni numeriche

Da un oggetto numerico si può ottenere un valore di un qualsiasi tipo predefinito numerico.

```
String sd = "123.45";
Double dD = new Double(sd); // 123.45;
int i = dD.intValue();  //123
double d = dD.doubleValue(); //123.45
float f = dD.floatValue(); //123.45
```

Metodi principali delle classi numeriche

```
byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()
```

Nota sulle conversioni

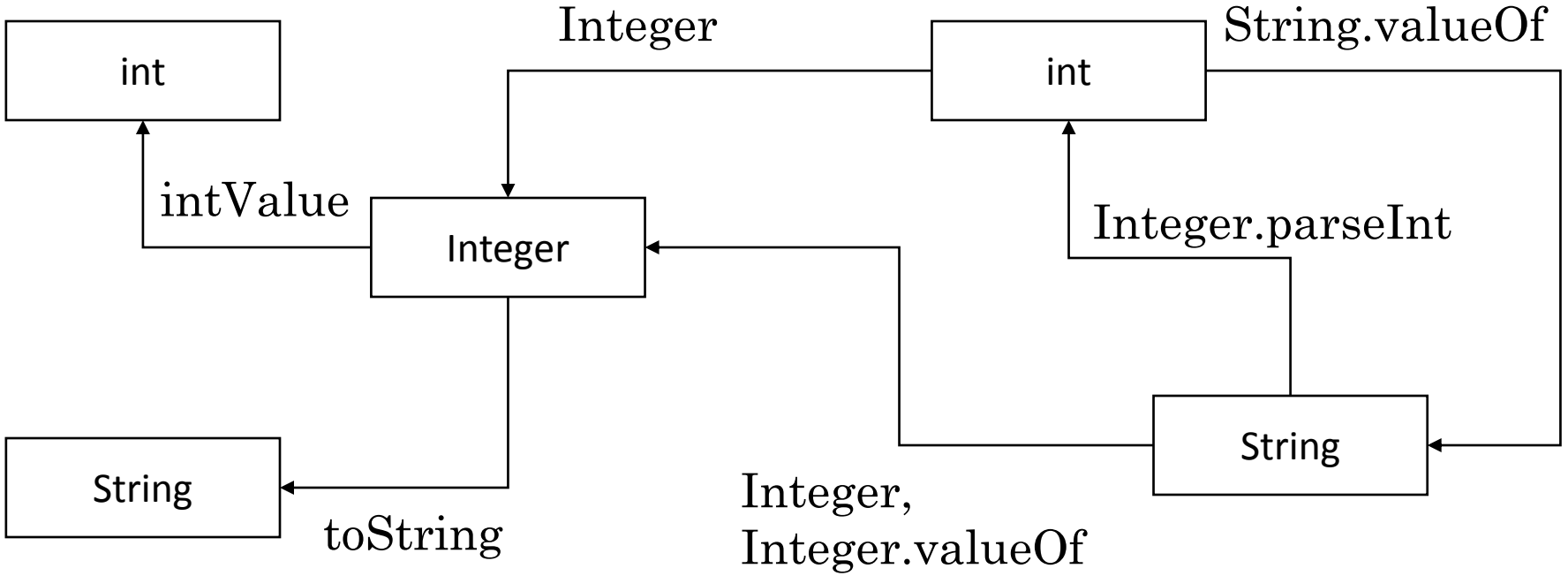
```
Float f = (float) 123.45;
Float f1 = dD.floatValue();
```

Costanti

Integer
public static final int **MIN_VALUE**
public static final int **MAX_VALUE**

Double
public static final double **MIN_VALUE**
public static final double **MAX_VALUE**

Conversioni (Integer, int, String)



Esercizio

```
public static void main(String[] args) {
        Integer j = 1;
        Integer k = incr (j + 1);
        System.out.println(k); // quanto vale k?
}

public static int incr(int i) {
        return i + 1;
}
```

Nota: nelle chiamate dei metodi il passaggio dei parametri avviene **by value**.

Array

Caratteristiche

Classe Arrays

Cursori (for compatto)

Metodo split di String

Ellissi

Array

Sequenze di elementi omogenei (valori oppure oggetti) con lunghezza predeterminata. Anche multidimensionali.

Es. di variabili array

```
int[] v;  // vettore di int
int[][] m;  // matrice di int
String[] sv;  // vettore di String
```

La classe **Arrays** (che si trova nel package java.util) fornisce utili metodi statici per la gestione degli array:

rappresentazione testuale: *toString*, *deepToString*

ordinamento: *sort*

Generazione

int[] a; // a è una variabile array di int, contenente null

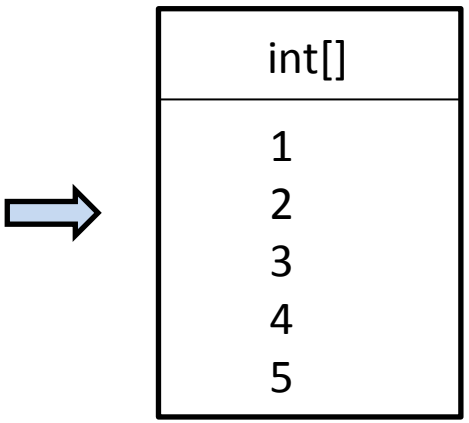
a = new int[10]; // a punta ad un "oggetto" array con 10 zeri.

int[] a = new int[10]; // forma compatta

Si può inizializzare l'array con un initializer:

int[] b = {1,2,3,4,5};

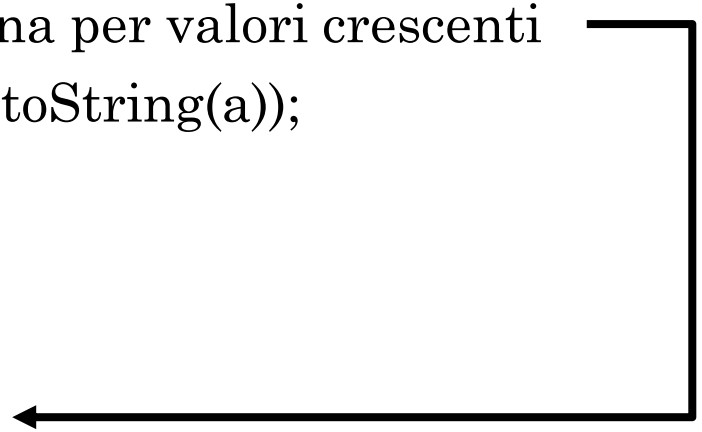
in questo caso è obbligatoria la forma compatta.




```
int[] a = new int[10];
for (int i = 0; i < a.length; i++) a[i] = i;
System.out.println(Arrays.toString(a));   // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sorting

```
a = new int[10];
for (int i = 0; i < a.length; i++) a[i] = 10 - i;
System.out.println(Arrays.toString(a));
Arrays.sort(a);  // ordina per valori crescenti
System.out.println(Arrays.toString(a));
```

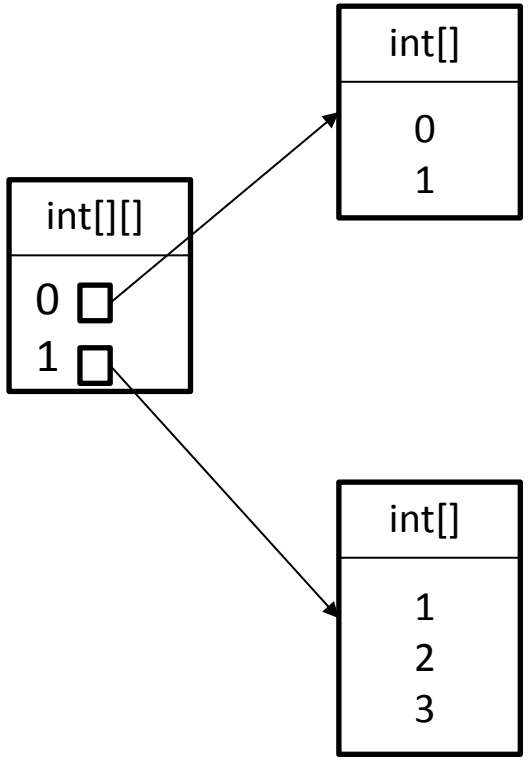


```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Array di array

```
int[][] m1 = new int[2][];
// la lungh. dell'array primario va specificata
for (int i = 0; i < m1.length; i++) {
        m1[i] = new int[i+2];
        for (int j = 0; j < m1[i].length; j++) {
            m1[i][j] = i + j;  }
}
System.out.println(Arrays.deepToString(m1));
//   [[0, 1], [1, 2, 3]]
```

Quanto vale m1[1][1]?

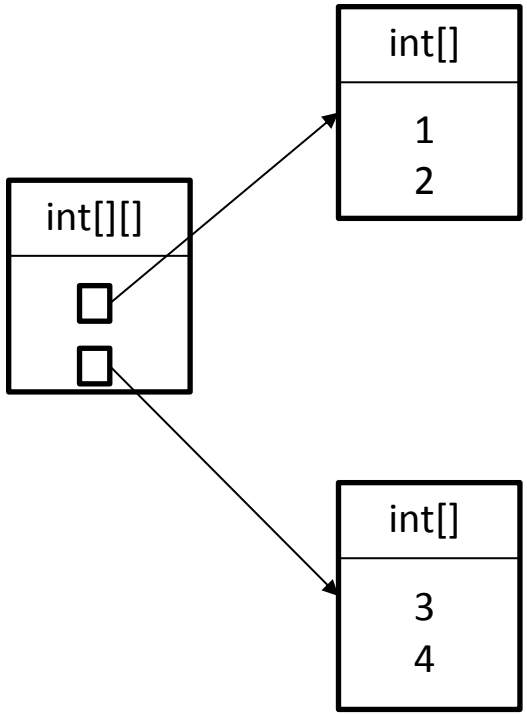


0 1
1 2 3

Array di array

```
int[][] m2 =  { {1,2}, {3,4} };  //initializer
System.out.println(Arrays.deepToString(m2));
// [[1, 2], [3, 4]]
```

```
1   2
3   4
```




```
int[][] m3 = new int[2][3];
System.out.println(Arrays.deepToString(m3));
//  [[0, 0, 0], [0, 0, 0]]
```

```
0  0  0
0  0  0
```

Con oggetti

```
String[] a1 = {"giallo", "rosso", "verde"};
String[][] m4 = { {"giallo", "rosso", "verde"}, {"alfa", "beta", "gamma"} };
String[][] m5 = new String [3][4]; // che cosa contiene?
```

for con cursore

Scrivere un vettore di interi e calcolare la somma degli elementi.

int [] vettore = {1,3,5,7};
int sum = 0; *for (int i:vettore)* sum += i;

i non è un indice ma un **cursore** che consente l'accesso a ciascun elemento dell'array anche in modifica

esempio:
for (int i:vettore) i++;

Esercizio

Scrivere un array di libri con codice int e titolo; indicare un titolo, trovare il libro corrispondente e stampare il codice.

Esercizio

Scrivere un array di libri con codice int e titolo per libro; indicare un titolo, trovare il libro corrispondente e stampare il codice.

```
public class Libro {
        private int codice; private String titolo;
        public Libro (int codice, String titolo) {
                this.codice = codice; this.titolo = titolo;}
        public int getCodice() {return codice;}
        public String getTitolo() {return titolo;}
}
```

Esercizio

nel main

```
Libro [] libri = {new Libro(1, "alfa"),new Libro(2, "beta"),
                  new Libro(3, "delta"),new Libro(4, "gamma")};

String titolo = "delta";
for (Libro l:libri) if (l.getTitolo().equals(titolo))  {
      System.out.println(l.getCodice());
      break;
}
```

Initializer

Forme equivalenti

String[] strings = {"alfa", "beta"};

String[] strings2 = new String[] {"alfa", "beta"};

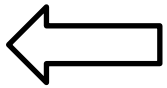
La seconda forma può risultare utile negli switch come mostrato nell'esempio seguente.

Domande d'esame

```
public class DomandeEsame {
final static public String[] questions = {
"Quali di queste affermazioni sono valide per un'interfaccia Java?", // Domanda 0
"Che cosa si può valutare di un test mediante un oracolo?" }; // Domanda 1
   final static public String[][] options = {
   { // Opzioni per la domanda 0
      "Un'interfaccia puo' essere vuota",
      "Un'interfaccia non puo' essere vuota",
      "Un'interfaccia deve contenere almeno un metodo astratto",
      "Un'interfaccia puo' contenere dei metodi astratti",
      "Un'interfaccia puo' contenere dei metodi statici",
      "Un'interfaccia non puo' contenere dei metodi statici",
   },{ // Opzioni per la domanda 1
      "la complessità del test",
      "la criticità dei test",
      "la correttezza dell'output",
      "la probabilità di fallimento",
      "la probabilità di successo"}};
```

Domande d'esame

```
/**
* Return the index of the right answer(s) for the given question
*/

public static int[] answer(int question){
        switch(question){
        case 0: return new int[]{0,3,4};
        case 1: return new int[]{2};
        }
        return null;
}
```



All'esame occorre indicare gli indici delle risposte scelte

Domande d'esame

```
public static void main(String[] args) {
        for(int q=0; q<questions.length; ++q){
            System.out.println("Question: " + questions[q]);
            int[] a = answer(q);
            if(a==null || a.length==0){
                System.out.println("<undefined>");
                continue;
            }
            System.out.println("Answer" + (a.length>1?"s":"") + ":" );
            for(int i=0; i<a.length; ++i){
                System.out.println(a[i] + " - " + options[q][a[i]]);
            }
        }
}
```

Question: Quali di queste affermazioni sono valide per un'interfaccia Java?
Answers:
0 - Un'interfaccia puo' essere vuota
3 - Un'interfaccia puo' contenere dei metodi astratti
4 - Un'interfaccia puo' contenere dei metodi statici
Question: Che cosa si può valutare di un test mediante un oracolo?
Answer:
2 - la correttezza dell'output

Suddivisione di una stringa in sottostringhe (split)

Metodo di String che suddivide la stringa data in sottostringhe raccolte in un array String[].

Il separatore delle sottostringhe può essere definito con un'espressione regolare.

```
String testo = "Se una notte d'inverno un viaggiatore";
String[] sottostringhe = testo.split(" "); // il separatore è uno spazio
System.out.println(Arrays.toString(sottostringhe));

// [Se, una, notte, d'inverno, un, viaggiatore]
```

Suddivisione di una stringa in sottostringhe (split)

```
testo = "Se, una, notte, d'inverno, un, viaggiatore";
 sottostringhe = testo.split(",\\s*");
// il separatore è una virgola eventualmente seguita da spazi
 System.out.println(Arrays.toString(sottostringhe)); // come sopra
```

L'argomento di split è un'espressione regolare:
",\\s*" indica virgola seguita da *n* spazi (*n*>=0).

invece \\s+ indica 1 o più spazi

Ellissi …

```
public class C {
public static int sum (int... val){ // tipo seguito da 3 punti
        int r = 0;
        for (int i:val) r+=i;
        return r;
}
// programma di test in un'altra classe
public static void main(String[] args) {
System.out.println(C.sum(10,20,30)); //60
System.out.println(C.sum()); //0
}}
```

Il tipo di val è int[]

Si può avere soltanto con l'ultimo parametro ed equivale ad un array di lunghezza non nota a priori (anche vuoto).

Con stringhe

Nel main

```
String s1 = "alfa";
String s2 = "beta";
String s3 = "gamma";
System.out.println(concatenazione(s1,s2,s3));// alfabetagamma

public static String concatenazione (String... stringhe) {
        StringBuilder sb = new StringBuilder();
        for (String s: stringhe) sb.append(s);
        return sb.toString();
}
```

Inheritance

Caratteristiche

Aspetti rilevanti:

- Polimorfismo
- Dynamic binding
- Down-cast
- Visibilità protected

Classi astratte

Classi e inheritance

Le classi sono organizzate ad albero con la classe (di libreria) **Object** come radice.

Ogni classe diversa da Object si dichiara erede (con la direttiva *extends*) di un'altra classe. Se manca extends, la classe è erede di Object.

Con l'inheritance, le proprietà di una classe sono ereditate (senza quindi bisogno di definirle) dalle classi eredi e ciò vale in modo transitivo.

Una classe può ridefinire i metodi ereditati che non siano **final**.

Frammento dell'albero delle classi

Object

- System
- String
- StringBuilder
- Number
  - Integer

Alcuni metodi di Object

```
public String toString();
public boolean equals(Object obj);
```

Ereditarietà

Una classe può ereditare le proprietà di un'altra classe, può ridefinirle e aggiungerne altre.

Aspetti rilevanti:

- Polimorfismo
- Dynamic binding
- Down-cast
- Visibilità protected

Esempio

Requisiti

Si rappresentino *particelle* come punti dotati di massa (valore intero).

La massa si può leggere e modificare.

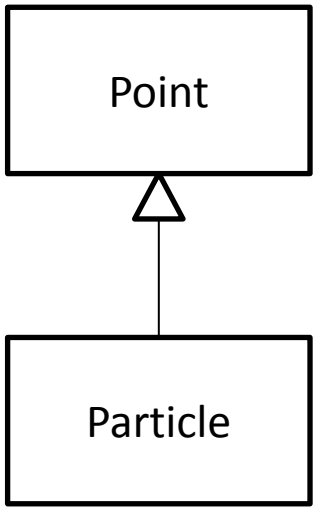
La particella fornisce una rappresentazione testuale che dà le coordinate e la massa; es. x = 1 y = 2 m = 100.

Soluzione

Si definisce la classe Particle come erede di Point.

```
Esempio d'uso
Particle pa1 = new Particle(10,20,100);
pa1.move(1,2); // metodo ereditato da Point
System.out.println(pa1); //  x = 1 y = 2 m = 100
```

Classe Particle



inheritance
(rappresentazione
grafica UML)

Particle è derivata (o sottoclasse) di Point.
Point è la superclasse diretta di Particle.

Che cosa serve in Particle?
l'attributo massa (m), con getter e setter
il costruttore
il metodo toString

Esempio d'uso
Particle pa1 = new Particle(10,20,100);
pa1.move(1,2); // metodo ereditato da Point
System.out.println(pa1); // x = 1 y = 2 m = 100

Classe Point (richiamo)

```
package p1;
public class Point {
    private int x,y; // attributi d'istanza
    public int getX() {return x;} // metodi getter
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;} // costruttore
    public void move(int x, int y) {this.x = x; this.y = y;} // metodo d'istanza
    public String toString(){return "x = " + x + " y = " + y;}
            // metodo d'istanza ereditato da Object e ridefinito
}
```

Particle

```
package p1;
public class Particle extends Point {
   private int m;
         public int getM() {return m;}
         public void setM(int m) {this.m = m;}
   public Particle(int x, int y, int m) {super(x,y); this.m = m;}
   public String toString(){return super.toString() + " m = " + m;}
}
```

Note

Logicamente un oggetto Particle include un oggetto Point; il costruttore di Particle deve quindi chiamare il costruttore di Point e ciò avviene con il **costrutto super()**.

Una particle vede tutti gli attributi non privati del point che contiene.

La classe Particle aggiunge la massa m e ridefinisce il metodo toString ereditato da Point.

L'implementazione del nuovo toString usa il toString dell'oggetto contenuto che indica con la **keyword super**.

Chiamata del costruttore di Point

public **Particle**(int x, int y, int m) {**super**(x,y); this.m = m;}

La prima istruzione nel costruttore di una classe derivata deve essere la chiamata di un costruttore della superclasse; se manca, il compilatore inserisce la chiamata del costruttore di default, **super()** o dà un errore se tale costruttore manca nella superclasse.

Siccome Point non ha il costruttore di default, scrivere

public **Particle**(int x, int y, int m) {this.m = m;}

è errato.

Programma di test

```
package p1;
public class Test {
public static void main(String[] args) {
Point p; Particle pa;
Point p1 = new Point(23, 94);
Particle pa1 = new Particle(10,20,100);
pa1.move(1,2); // metodo ereditato
System.out.println(pa1); // chiama pa1.toString()  x = 1 y = 2 m = 100

p = p1; // p punta ad un point
System.out.println(p); // chiama toString() di Point x = 23 y = 94

p = pa1; // p punta ad una Particle (up-cast)
System.out.println(p); // chiama toString() di Particle   x = 1 y = 2 m = 100
```

Note

p = p1; // p punta ad un point
System.out.println(p); // chiama toString() di Point x = 23 y = 94
p = pa1; // p punta ad una Particle
System.out.println(p); // chiama toString() di Particle x = 1 y = 2 m = 100

polimorfismo: prima p punta ad un Point, poi ad una Particle;
p è definito di tipo Point, ma può puntare anche ad un oggetto di una classe derivata da Point;
dynamic binding: la versione di toString() da chiamare dipende dal tipo dell'oggetto effettivamente puntato.

down-cast

p.setM(10);

errore rilevato dal compilatore: p è definito di tipo Point, quindi non vede setM

pa = p;

inaccettabile; se pa puntasse ad un Point la chiamata di pa.setM lecita sintatticamente, produrrebbe un errore a run-time

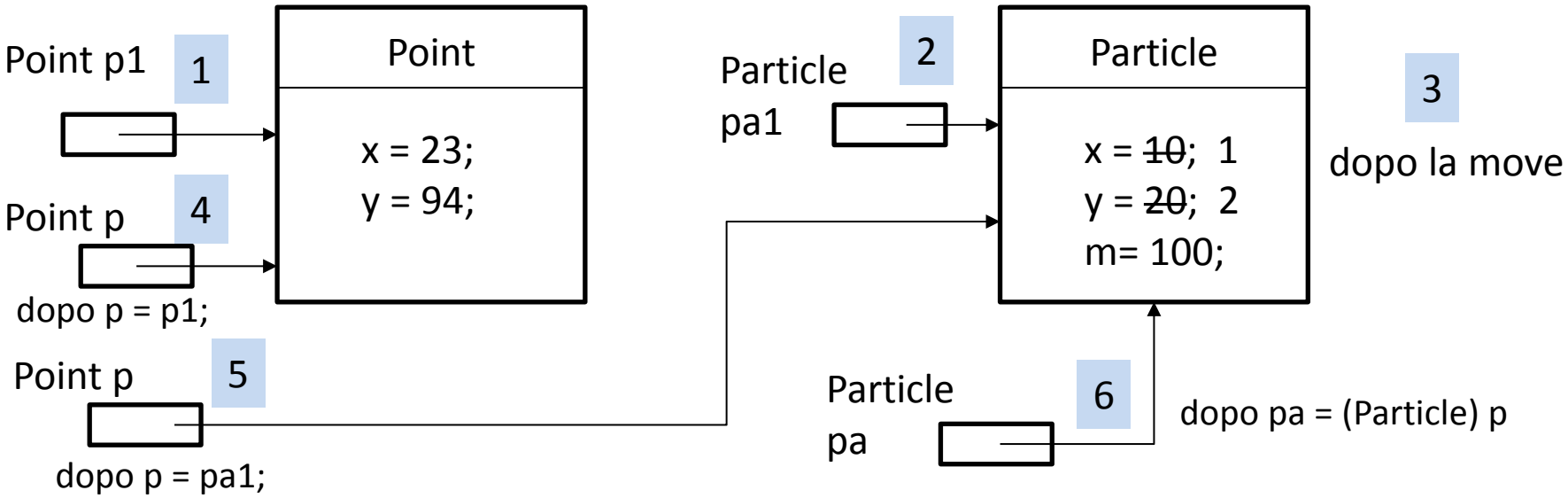
pa = (Particle) p; **down-cast, accettabile sintatticamente ma**

dà errore se p non punta ad una Particle (o ad un oggetto derivato da Particle)

pa.setM(10); operazione corretta perché pa punta ad una Particle

In sintesi

1. Point p1 = new Point(23, 94);
2. Particle pa1 = new Particle(10, 20, 100);
3. pa1.move(1,2); // metodo ereditato
4. p = p1; // p punta ad un point
5. p = pa1; // p punta ad una Particle
6. pa = (Particle) p



Livelli di visibilità

Livelli di visibilità delle proprietà di una classe

1. **public**: visibile da tutte le altre classi
2. **private**: visibile nella classe corrente
3. - assenza di qualificatore (*livello package*): visibile dalle altri classi dello stesso package
4. **protected**: visibile dalle classi derivate e dalle altre classi dello stesso package.

Livello protected

Esempio

I metodi toString di Point e Particle devono scrivere il tipo dell'oggetto, come

```
point: x = 23 y = 94
particle: x = 10 y = 20 m = 100
```

La soluzione è basata sull'attributo **String info = "point";**
definito nella classe Point e ridefinibile in Particle.

Che livello di visibilità deve avere info in Point?

Dipende dalla collocazione delle due classi.

Programma di test

```
p = new Point(23, 94); System.out.println(p);
Particle pa = new Particle(10,20,100); System.out.println(pa);
```

Scenario 1

Nello stesso package Point e Particle

```
public class Point {
String info = "point";
        private int x,y; public int getX() {return x;} public int getY() {return y;}
        public Point(int x, int y) {this.x = x; this.y = y;}
        public void move(int x, int y) {this.x = x; this.y = y;}
public String toString(){return info + ": x = " + x + " y = " + y;}
}
```

info ha visibilità di livello package e quindi Particle può ridefinirlo

```
public class Particle extends Point {
   private int m;    public int getM() {return m;}    public void setM(int m) {this.m = m;}
   public Particle(int x, int y, int m) {
        super(x,y); this.m = m; info = "particle";}
   public String toString(){return super.toString() + " m = " + m;}
}
```

Scenario 2

Se Point e Particle non si trovano nello stesso package

info deve avere il livello di visibilità *protected*.

```
public class Point {
    protected String info = "point";
    private int x,y;
    public int getX() {return x;}
    public int getY() {return y;}
    public Point(int x, int y) {this.x = x; this.y = y;}
    public void move(int x, int y) {this.x = x; this.y = y;}
    public String toString(){return info + ": x = " + x + " y = " + y;}
}
public class Particle extends Point {  // come nel caso precedente
```

Array con oggetti di classi diverse

Un array di tipo T accetta anche oggetti di classi derivate da T così come una variabile di tipo T può puntare ad un oggetto di una classe derivata da T (polimorfismo).

```
Point[] v = {new Point(10,20), new Particle(1,2,100), new Point(0,1)};
for (Point point:v){System.out.println(point);}
```

```
x = 10 y = 20
x = 1 y = 2 m = 100
x = 0 y = 1
```

Qualificatore final

Una classe final (come String) non può avere sottoclassi.

Una variabile final si comporta come una costante.
Es. final int i = 10; // i non è più modificabile

Un metodo final non può essere ridefinito.

Classi astratte

Contengono almeno un metodo astratto (**abstract**), cioè privo di body.

Possono anche contenere soltanto metodi astratti.

Una classe astratta impone l'implementazione di certe funzionalità (i metodi astratti) alle sue sottoclassi. Se l'implementazione è parziale la sottoclasse rimane astratta.

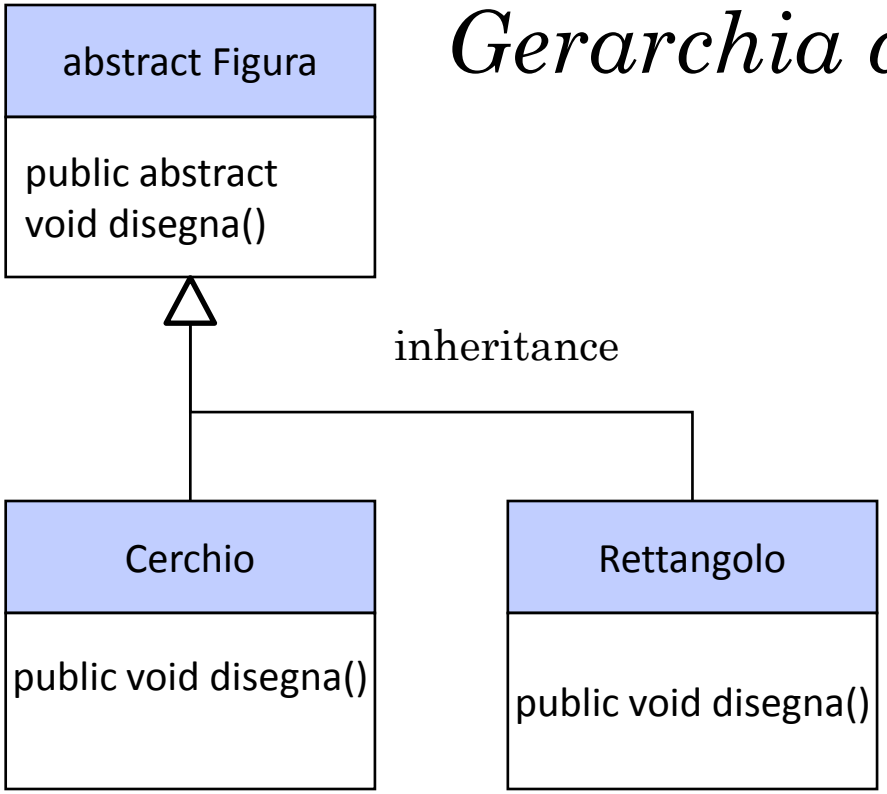
Class Number

```
public abstract class Number extends Object {

public abstract  double doubleValue();
public abstract  float floatValue();
public abstract  int intValue();
public abstract  long longValue();
public byte byteValue();
public short shortValue()

}
```

Sottoclassi principali
Double, Float, Integer, Long, Byte, Short

possono convertire il proprio valore negli altri formati.

Gerarchia di classi



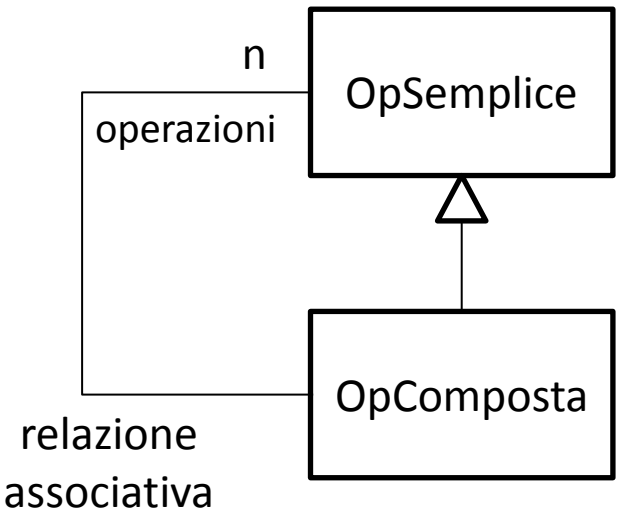
Tutte le figure devono essere disegnabili.

```
public static void test() {
Figura[] v = {new Cerchio(), new Rettangolo()};
for (Figura f: v){ (f.disegna();}}
```

Esercizio su inheritance

Calcolare la durata di un'operazione composta.

Un'operazione composta comprende un certo numero di operazioni semplici e composte. Un'operazione semplice ha una *durata elementare* (es. 10 unità di tempo). Un'operazione composta ha una *durata* data dalla somma delle durate delle operazioni componenti + la sua durata elementare.



attributi: int durataE
metodi
OpSemplice(int durataE)
int durata()

attributi: OpSemplice[] operazioni
metodi
OpComposta(int durataE, OpSemplice… operazioni)
int durata() // ridefinizione

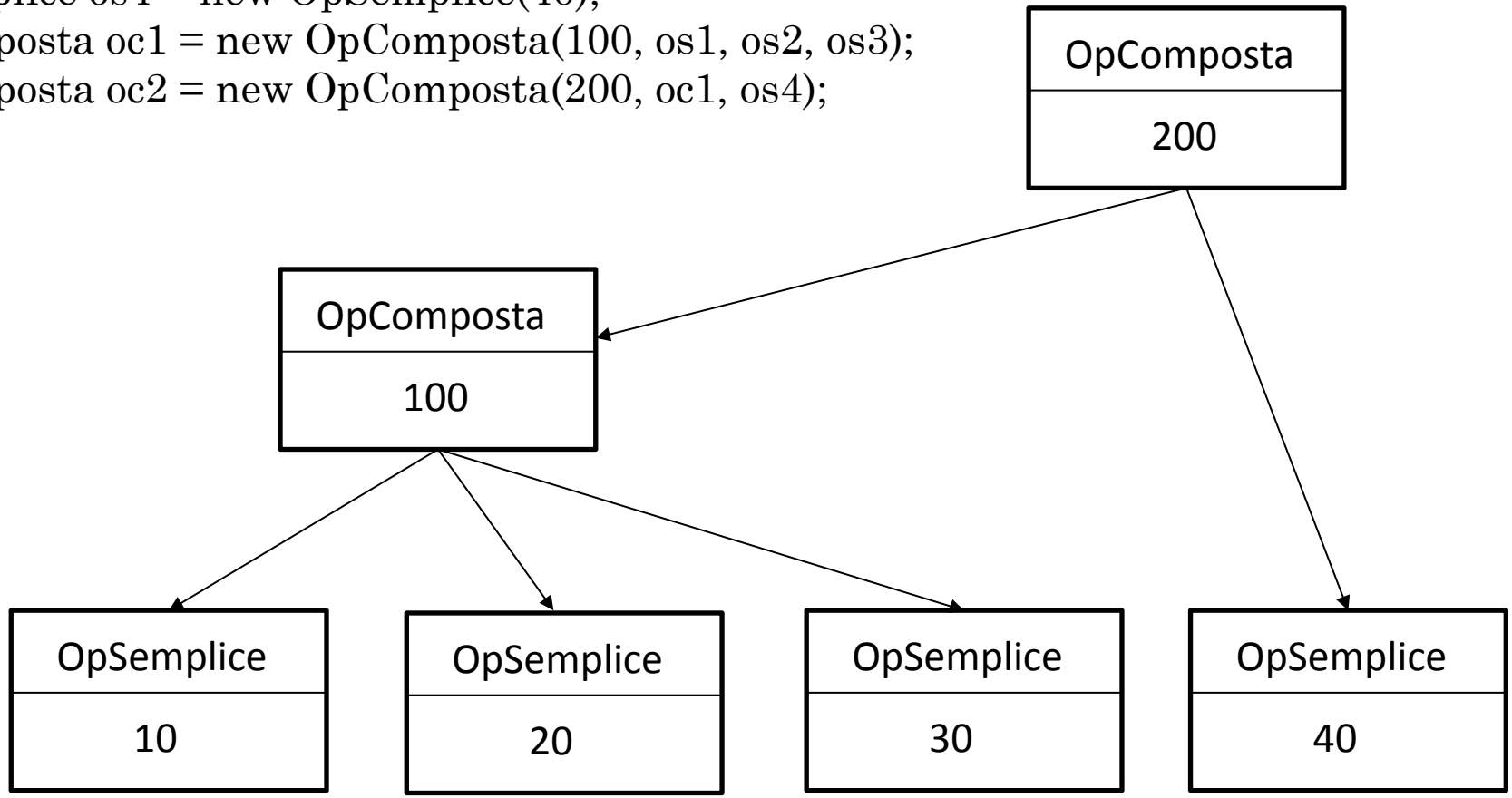
Esercizio su inheritance

Es. d'uso

```
public class TestOperazioni {
public static void main(String[] args) {
OpSemplice os1 = new OpSemplice(10);
OpSemplice os2 = new OpSemplice(20);
OpSemplice os3 = new OpSemplice(30);
OpSemplice os4 = new OpSemplice(40);
OpComposta oc1 = new OpComposta(100, os1, os2, os3);
OpComposta oc2 = new OpComposta(200, oc1, os4);
System.out.println(oc2.durataComplessiva()); //400
}}
```

Modello a oggetti

```
OpSemplice os1 = new OpSemplice(10);
OpSemplice os2 = new OpSemplice(20);
OpSemplice os3 = new OpSemplice(30);
OpSemplice os4 = new OpSemplice(40);
OpComposta oc1 = new OpComposta(100, os1, os2, os3);
OpComposta oc2 = new OpComposta(200, oc1, os4);
```



Esercizio

```
public class OpSemplice {
int durataE;
public  OpSemplice (int durataE) {this.durataE = durataE;}
public int durata () {return durataE;}
}

public class OpComposta extends OpSemplice {
OpSemplice [] operazioni;
public OpComposta (int durataE, OpSemplice ... operazioni) {
        super(durataE); this.operazioni = operazioni;}
public int durata ()  // ridefinita
        {int d = durataE;
        for (OpSemplice op: operazioni) d += op.durata();
        return d;}
}
```

Eccezioni

Significato

Eccezioni runtime ed eccezioni checked

Classi di eccezioni

Trattamento: blocchi try e catch

Lancio: throw

Eccezioni

Se l'interprete durante l'esecuzione di un metodo trova un'istruzione errata, interrompe il metodo e lancia un oggetto che rappresenta l'eccezione trovata: ad es. divisione per 0, accesso a un array con un indice fuori range, accesso ad un oggetto con un riferimento null.

Es. di codice

```
int x = 0; int a = 10;
int b = a/x;                    java.lang.ArithmeticException: / by zero

int[] v = {100, 200, 300};
System.out.println(v[x-1]);  java.lang.ArrayIndexOutOfBoundsException: -1

String s = "alfa";
s = null;
System.out.println(s.length()); java.lang.NullPointerException
```

Queste eccezioni sono dette di **runtime**.

Eccezioni checked

Il programmatore può decidere di lanciare un'eccezione se i dati su cui il programma deve operare risultano errati.

Ad es.
se l'istruzione a * b /100 richiede che a e b siano >= 0

il programmatore può inserire un controllo:

```
if (a < 0 || b < 0) throw new Exception("dati errati");
return a * b /100;
```

In caso di errore l'interprete lancia un oggetto di tipo Exception con la causale "dati errati".

Queste eccezioni sono dette **checked**.

Eccezioni

Quando si rileva un problema nell'esecuzione di un metodo, è lanciata un'eccezione, che, se non è trattata nel metodo stesso o in un metodo chiamante, interrompe il programma.

Il problema è rilevato dall'interprete java (ad es. una divisione per zero) oppure dal programma (ad es. un'incongruenza dei dati).

Casi principali di eccezione

Tipi di eccezione predefiniti

Tipi definiti dal programmatore

Gestione delle eccezioni

- try catch
- finally
- throw e throws

Eccezioni (3 gruppi)

errori: non dipendono dal programma e non sono trattabili

classe Error e derivate;

checked exceptions: dichiarate dai metodi che possono lanciarle

classe Exception e derivate escluse RuntimeException e derivate;

runtime exceptions: non dichiarate e solitamente rilevate dal runtime system

classe RuntimeException e derivate.

Le eccezioni sono oggetti le cui classi si collocano in una zona precisa dell'albero delle classi.

Classi di eccezioni

Object

Throwable

Error

--- sottoclassi di Error

Exception

--- classi di checked exceptions

RuntimeException

--- sottoclassi di RuntimeException

Esempi di sottoclassi di RuntimeException

ArithmeticException

IndexOutOfBoundsException

NullPointerException

IllegalArgumentException - NumberFormatException

Esempio di runtime exception

```
public class Esempio1 {
public static void main(String[] args) {
        int i = Integer.parseInt(args[0]);
        System.out.println(i);
}}
```

Il programma cerca di convertire la stringa di input (args[0]) in un valore intero.

Il metodo parseInt lancia un'eccezione di tipo ArrayIndexOutOfBoundsException se non c'è nessun input

oppure un'eccezione di tipo NumberFormatException se il contenuto non corrisponde ad un numero (es. 10a),

e interrompe il programma.

try/catch

Per trattare le eccezioni che una singola istruzione o un blocco di istruzioni possono lanciare, si usa un blocco try/catch. Le catch sono exception handlers.

```
public class Esempio2 {
public static void main(String[] args) {
int i = 0; String failureMsg = null;
try {
        i = Integer.parseInt(args[0]);
} catch (ArrayIndexOutOfBoundsException e) {
        failureMsg ="no input";
} catch (NumberFormatException e ) {
        failureMsg ="wrong input";
}

if (failureMsg != null) System.out.println(failureMsg);
else System.out.println(i);}}
```

Il blocco try contiene il codice (una o più linee) che può lanciare un'eccezione.
L'eccezione è catturata da una catch; solitamente, dopo l'esecuzione di una catch il programma continua con le istruzioni che seguono il blocco try/catch.

Checked exceptions

La classe di una checked exception è Exception o una derivata di Exception escluse RuntimeException e le sue derivate.

Se un metodo può lanciare una checked exception deve dichiararlo nell'intestazione con la clausola *throws*.

Il metodo seguente lancia un'eccezione se gli input sono negativi.

```
public class Esempio3 {

static double calcoloPercentuale(double a, double b) throws Exception {
        if (a < 0 || b < 0) throw new Exception("dati errati");
        return a * b /100;
}
```

Programma di test

```
public static void main(String[] args) {
try {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        System.out.println(calcoloPercentuale(a,b));
} catch (RuntimeException e)  {
        System.out.println("input errati");
} catch (Exception e)  {
        System.out.println(e.getMessage());
} finally {System.out.println("end");}
}
```

quale ordine nelle catch?

precedenze tra 2 catch:
se B è derivata da A
l'ordine è
catch B
catch A

Se fosse la prima, la catch A servirebbe sia l'eccezione A sia la B perché un B è anche un A ma non viceversa.

L'ordine delle due catch è importante: perché?

Il blocco finally viene comunque eseguito.

Il compilatore forza l'inserimento della seconda catch, perché Exception è un'eccezione controllata (checked exception).

Variante nel programma di test

```
public static void main(String[] args) throws Exception {
try {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        System.out.println(calcoloPercentuale(a,b));
} catch (RuntimeException e)  {
        System.out.println("input errati");
} finally {System.out.println("end");}
}
```

Il main non tratta l'eccezione (Exception) sollevata dal metodo ma la lascia passare (e deve dichiararlo!).

Note

I blocchi try/catch possono essere annidati.

Una catch può rilanciare un'eccezione (quella che ha ricevuto oppure un'altra).

Classe Object

Lista dei metodi

Analisi di equals, hashcode e toString

I metodi della classe Object

public String toString()
public boolean equals(Object obj) // confronti in base al contenuto
public int hashCode() // mapping da un oggetto a int
protected Object clone() throws CloneNotSupportedException
public final Class<?> getClass()
protected void finalize() throws Throwable
// per la gestione dei thread
public final void notify()
public final void notifyAll()
public final void wait()   throws InterruptedException

Due punti sono uguali se hanno uguali entrambe le coordinate (metodo equals)

```
public class Point {
   private int x,y;  public int getX() {return x;} public int getY() {return y;}
            public Point(int x, int y) {this.x = x; this.y = y;}
            public void move(int x, int y) {this.x = x; this.y = y;}
            public String toString(){return "x = " + x + " y = " + y;}
   public boolean equals (Object o) { // usato nelle collezioni
            if (!(o instanceof Point)) return false;
            Point p = (Point) o; return this.x == p.x && this.y == p.y;}

   public boolean isEqualTo(Point p) { // più specifico ma ignorato dalle collezioni
            return this.x == p.x && this.y == p.y;}}
```

Da notare:

- l'operatore **instanceof**: object instanceof className dà true se la classe dell'oggetto puntato dal riferimento object è className
- se o punta ad un Point occorre fare un down-cast ad un riferimento di tipo Point per poter confrontare le coordinate.

Programma di test

```
public static void main(String[] args) {
   Point p1 = new Point(23, 94);
   p1.move(10,20);
   Point p2 = new Point(10, 20);
   System.out.println(p1.equals(p2)); // true
   System.out.println(p1 == p2); // false
   System.out.println(p1.equals("alfa")); // false
   System.out.println(p1.isEqualTo(p2)); // true
}
```

hashCode e toString

```
package esempiPrimaParte;
public class Info {
        int val;
        public Info(int val) {this.val = val;}
}

Programma di test
public static void main(String[] args) {
        Info a = new Info(10);
        Info b = new Info(10);
        System.out.println(a.hashCode());
        System.out.println(b.hashCode());
        System.out.println(a);
        System.out.println(b);
}
}
```

Per evitare duplicazioni le collezioni usano il metodo hashCode che deve quindi dare lo stesso valore per oggetti "uguali". Se due oggetti hanno hashCode diversi sono certamente diversi, altrimenti si fa un test più accurato con il metodo equals.

```
31168322
17225372
esempiPrimaParte.Info@1db9742
esempiPrimaParte.Info@106d69c
```

Ridefinizione di hashCode e toString

```
package esempiPrimaParte;
public class Info1 {
            int val;
public Info1(int val) {this.val = val;}
public int hashCode () {return val;}
public String toString () {return String.valueOf(val);}

public static void main(String[] args) {
Info1 a = new Info1(10);
Info1 b = new Info1(10);
System.out.println(a.hashCode());
System.out.println(b.hashCode());
System.out.println(a);
System.out.println(b);
}
}
```

```
10
10
10
10
```

2 int e 2 stringhe

Tipi enumerativi (enum)

Significato

Esempi

Tipi enumerativi (enum)

Un tipo enumerativo definisce una classe e anche tutti i suoi oggetti.

Esempi

Tipo enumerativo per i mesi dell'anno.

Tipo enumerativo per i mesi dell'anno con la durata standard per mese (28 giorni per febbraio).

Il metodo statico **values** fornisce la collezione degli oggetti del tipo enumerativo.

Enumerativi semplici (senza attributi)

```
public enum Mese0 {
GENNAIO,FEBBRAIO,MARZO,APRILE,MAGGIO,GIUGNO,
LUGLIO,AGOSTO,SETTEMBRE,OTTOBRE,NOVEMBRE,DICEMBRE
}

programma di test
public static void main(String[] args) {
for (Mese m: Mese.values()) System.out.println(m);
// toString stampa il nome dell'oggetto
}
```

```
GENNAIO
FEBBRAIO
…
```

Enumerativi (con attributi)

```
public enum Mese1 {
GENNAIO (31), FEBBRAIO (28), MARZO (31), APRILE (30), MAGGIO (31),
GIUGNO (30), LUGLIO (31), AGOSTO (31), SETTEMBRE (30),
OTTOBRE (31), NOVEMBRE (30), DICEMBRE (31);
        private int nGiorni;
        Mese1(int nGiorni) {this.nGiorni = nGiorni;}
        public int nGiorni() {return nGiorni;}
}

programma di test
public static void main(String[] args) {
for (Mese1 m:Mese1.values()) System.out.println(m + " " + m.nGiorni());
}
```

```
GENNAIO 31
FEBBRAIO 28
…
```

Vario

Classi Math e System

Garbage collection

Classi Math e System

Math è una collezione di metodi (statici) matematici.

System fornisce i riferimenti agli standard stream (in, out, err),
e vari metodi, tra cui il garbage collector.

Alcuni metodi di Math

```
static double   abs(double a)
static double   cos(double a)
static double   exp(double a)
static double   log(double a)
static double   max(double a, double b)
static double   random() // tra 0 e 1 compresi
static long     round(double a)
static double   sqrt(double a)
```

Esempio: fornire una valore uniforme tra 2 long, l e h (l <= h)

```
public static long uniform(long l, long h) {
        double d = Math.random() ;
        return Math.round(l + d * (h - l));        }
```

Alcuni metodi di System

static void **arraycopy** (Object src, int srcPos, Object dest, int destPos, int length)
static long **currentTimeMillis**()
static void exit(int status)
static void **gc**() // esegue la garbage collection
static void runFinalization()

Esempio di arraycopy

```
public static void main(String[] args) {
        String[] s1 =  {"alfa", "beta", "gamma", "lambda", "zeta", "omega"};
        String[] s2 = new String[3];
        System.arraycopy(s1, 2, s2, 0, 3);
        // primo array, indice del primo array,
        // secondo array, indice del secondo array, n. di elementi da copiare
        System.out.println(Arrays.toString(s2)); //[gamma, lambda, zeta]
}
```

Garbage collection

Gli oggetti hanno un contatore dei riferimenti attivi. Quando il contatore di un oggetto scende a 0, l'oggetto non è più accessibile (si è perso ogni riferimento).

Periodicamente, in modo automatico, il runtime di java libera la memoria della heap occupata dagli oggetti inaccessibili; con il metodo **gc** si può chiedere esplicitamente il recupero della memoria inutilizzata.

Prima della distruzione di un oggetto è chiamato il metodo *finalize*, che consente di programmare il rilascio di risorse (esterne) non trattate dal garbage collector.

Esempio di oggetto inaccessibile:

```
Point p1 = new Point(10,20);

p1 = null;
```