Introduzione a Java
3

Giovanni Pardini
pardinig@di.unipi.it

Dipartimento di Informatica
Università di Pisa

13 dicembre 2009

Sommario

Sommario

Gerarchia di tipi

Tutte le classi definite in Java sono sottoclassi (dirette o indirette) della classe speciale `Object`

- la relazione di sottoclasse è transitiva

Ogni classe che non ne estende espicitamente un'altra (keyword `extends`), viene definita implicitamente come sottoclasse di `Object`

Le seguenti dichiarazioni di classe sono equivalenti:

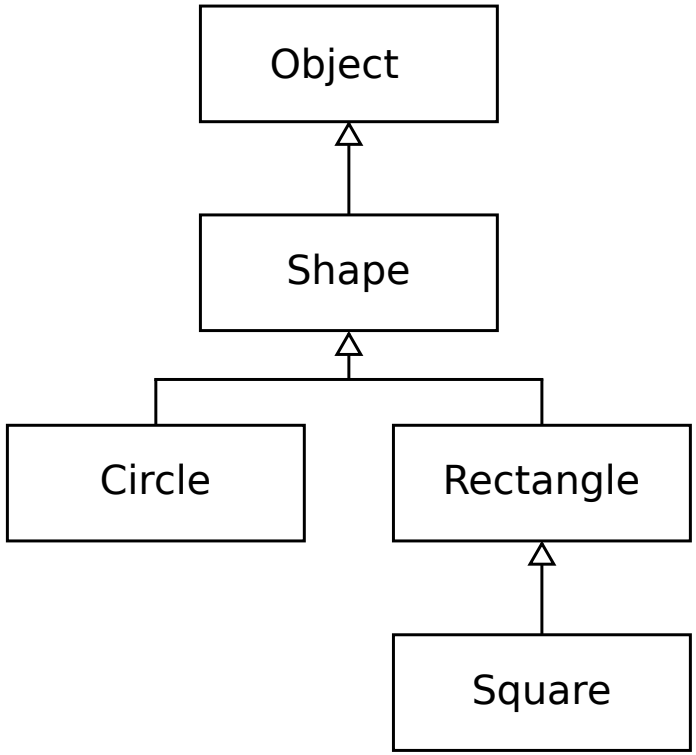
```
public class <NomeClasse> {
  ...
}

public class <NomeClasse> extends Object {
  ...
}
```

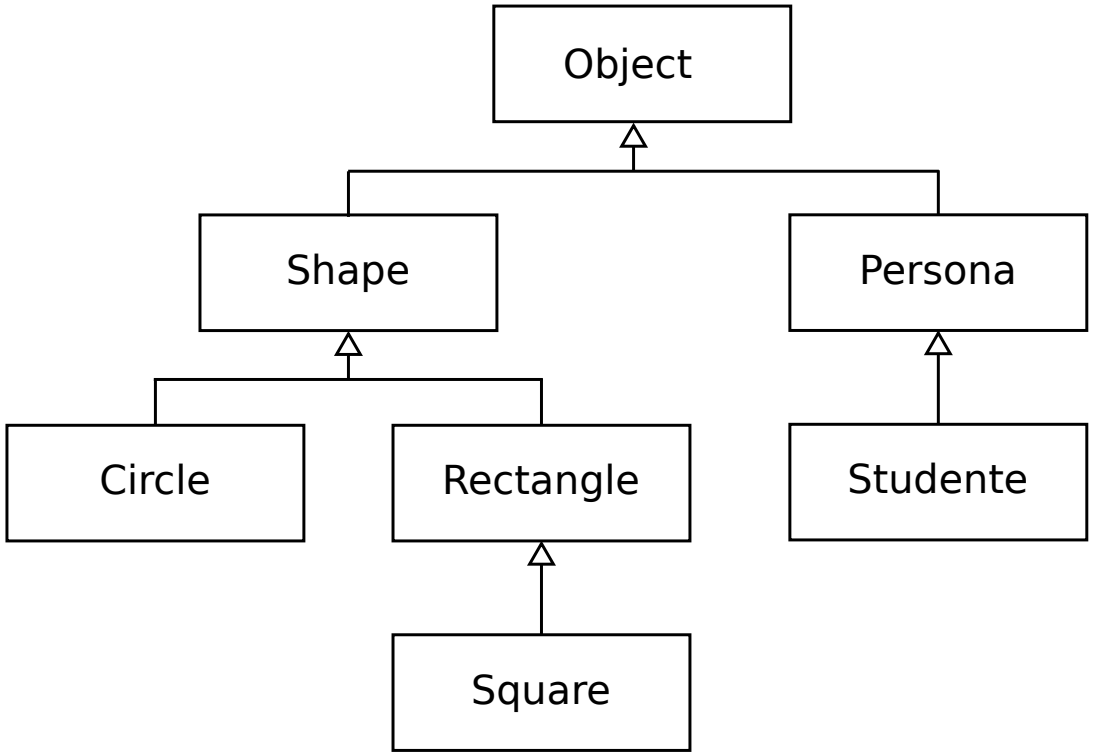
Si crea una gerarchia di classi (o di tipi) con in cima la classe `Object`

- tutte le altre classi sono sottotipi di `Object`

Gerarchia di tipi: esempio (1)



Gerarchia di tipi: esempio (2)



Principio di sostituzione

Principio di sostituzione

Un oggetto del sottotipo può essere utilizzato dovunque sia richiesto un oggetto del supertipo.

Il principio di sostituzione per le classi è soddisfatto grazie all'ereditarietà

- gli oggetti della sottoclasse hanno almeno tutte le variabili d'istanza ed i metodi della superclasse
- la sottoclasse può definire altri metodi e variabili

- In generale, è utile che anche il comportamento delle sottoclassi sia "corretto" rispetto a quello atteso dalla superclasse

Principio di sostituzione: esempio (1)

Un'istanza di `Studente` si può usare in qualsiasi espressione dove sia richiesto un oggetto di `Persona`

- assegnamento, passaggio dei parametri

Esempio

La classe `Persona` definisce un metodo d'istanza con firma

```
boolean omonimo(Persona p1)
```

che può essere usato anche con oggetti di tipo `Studente`

```
1  Persona tizio = new Studente("Mario", "Lucca"); // assegn.
2  tizio.visualizza();      // invocazione di un metodo
3                           // ereditato da Persona
4
5  Studente caio = new Studente("Mario", "Pisa");
6  if (tizio.omonimo(caio))   // ok: il metodo richiede
7      ...                    // un parametro di tipo Persona
```

Principio di sostituzione: esempio (2)

Attenzione: non è possibile il contrario, ovvero utilizzare un oggetto del supertipo al posto di uno del sottotipo

- il sottotipo può avere variabili e metodi aggiuntivi rispetto al supertipo

Esempio

```
1  Persona tizio = new Persona("Pluto", "Lucca");
2  Studente caio = tizio;  // errore di tipo
3
4  String s = tizio.getPdS();  // errore: la classe Persona non
5                              // ha un metodo getPdS()
```

- Questo errori sono rilevati staticamente dal compilatore

Tipo apparente/effettivo

Per una variabile di tipo oggetto, si distingue tra tipo apparente ed effettivo

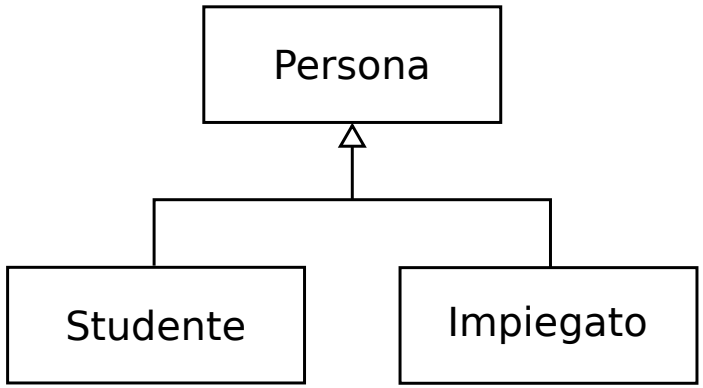
- tipo apparente: tipo con cui la variabile è dichiarata
- tipo effettivo: tipo dell'oggetto riferito dalla variabile

Il *tipo apparente* è conosciuto a tempo di compilazione e non cambia, mentre il *tipo effettivo* può variare durante l'esecuzione

- Per il principio di sostituzione, se la variabile non è `null`, ad ogni passo durante l'esecuzione si hanno due sole possibilità:
  - il tipo effettivo è uguale al tipo apparente *oppure*
  - il tipo effettivo è un sottotipo del tipo apparente

Tipo apparente/effettivo: esempio

Si supponga di avere una classe `Impiegato`, sottoclasse di `Persona`




```
1  Persona tizio = null;  // tipo effettivo: nessuno
2
3  tizio = new Studente("Mario Rossi", "Pisa");
4                         // tipo effettivo: Studente
5
6  tizio = new Impiegato("Mario Rossi", "Lucca", "Poste");
7                         // tipo effettivo: Impiegato
```

- La variabile `tizio` ha tipo apparente `Persona`

Controllo dei tipi

Il compilatore Java effettua staticamente (a tempo di compilazione) un controllo dei tipi

Il compilatore di Java effettua una analisi statica sul programma, controllando che ogni utilizzo delle variabili sia corretto rispetto ai tipi apparenti

- i controlli sui tipi effettivi sono svolti invece a tempo di esecuzione

- Il compilatore non potrebbe controllare i tipi effettivi
  - sono conosciuti solo durante l'esecuzione (dipendono dal flusso di esecuzione)

Controllo dei tipi: esempio

```
1 Persona tizio = new Studente("Mario Rossi", "Pisa");
2 tizio.setPdS(Algebra);    // errore: la classe Persona non
3                           // ha un metodo setPdS()
```

Il compilatore controlla il tipo apparente `Persona`

- il tipo effettivo sarebbe giusto, ma il compilatore non lo conosce!

Cast (1)

L'operazione di casting permette di modificare il tipo apparente (del valore) di una variabile

Nel caso delle classi, il casting permette di considerare (il valore di) una variabile come appartenente ad un sottotipo del suo attuale tipo apparente

```
(<SottoClasse>) <oggetto>
```

Esempio

```
1 Persona tizio = new Studente("Mario Rossi", "Pisa");
2 Studente s = (Studente)tizio;  // cast a Studente
3 s.setPdS("Algebra");  // ok
```

Terminologia

- upcast: cast (implicito) verso un *supertipo*
- downcast: cast (esplicito) verso un *sottotipo*

Cast (2)

Il cast è possibile solo verso sottotipi dell'attuale tipo apparente

- controllo statico del compilatore

In ogni caso, a run-time, viene sempre controllato che il tipo effettivo sia sottotipo di quello apparente

- anche se c'è un cast

Esempio

```
1 Persona tizio = new Studente("Mario Rossi", "Pisa");
2 Shape s = (Shape)tizio;  // errore: Shape non e'
3                          // sottotipo di Persona
```

Esempio

```
1 Persona tizio = new Studente("Mario Rossi", "Pisa");
2 Impiegato p1 = (Impiegato)tizio;  // errore a run-time:
3                   // Studente non e' sottotipo di Impiegato
```

Controllo del tipo a run-time

L'operatore `instanceof` permette di controllare, a run-time, se il tipo effettivo è sottotipo della classe specificata

```
<oggetto> instanceof <NomeClasse>
```

- si ottiene un valore `boolean`, che vale `true` se e solo se l'oggetto è istanza della classe

Esempio

```
1 if (tizio instanceof Studente) {
2   ((Studente)tizio).setPdS("Algebra");  // ok
3 }
```

- In questo caso, il cast non può mai fallire

Late binding

Le sottoclassi possono ridefinire (*override*) metodi d'istanza definiti nelle superclassi

In Java, invocando un metodo d'istanza overridden, viene sempre eseguito il metodo più specifico

- dipende dal tipo effettivo

- La scelta del metodo da eseguire viene fatta a run-time (*late binding*)
  - si risale la gerarchia dei tipi a partire dal tipo effettivo dell'oggetto verso la radice `Object`,
  - la prima definizione del metodo richiesto che si trova (la più specifica) viene eseguita

Attenzione: si ha late binding sempre e **solo** per i metodi d'istanza!

Late binding: esempio

Il metodo `visualizza()` è riscritto nella sottoclasse `Studente`

- stampa, oltre a `nome` e `indirizzo` (ereditati), anche il contenuto delle variabili `matricola` e `pianoDiStudio`

```
1 Studente tizio = new Studente("Mario Rossi", "Pisa");
2 tizion.setPdS("Algebra");
3 Persona p1 = tizio; // upcast
4 p1.visualizza();    // esegue il metodo di Studente
```

- La chiamata al metodo `visualizza()` esegue il metodo overriden più specifico, a partire dal tipo effettivo `Studente`
  - appunto, quello della classe `Studente`

Late binding e risoluzione dei nomi

In Java, si ha late binding solo (e sempre) per i metodi d'istanza

Per tutti gli altri elementi si ha *early binding*

- i nomi vengono risolti staticamente dal compilatore
- variabili d'istanza e statiche
- metodi statici

Late binding e risoluzione dei nomi: esempio (1)

```
1 public class A {  // definizione classe A
2   String getNome() {
3     return "A";
4   }
5
6   static String getNomeStatico() {
7     return "A";
8   }
9 }
```

```
1 public class B extends A { // definizione sottoclasse B di A
2   String getNome() {
3     return "B";
4   }
5
6   static String getNomeStatico() {
7     return "B";
8   }
9 }
```

Late binding e risoluzione dei nomi: esempio (2)

```
1  public class Test {
2    public static void main(String[] args) {
3      B b = new B();
4      A a = b;
5
6      // chiamata metodi d'istanza
7      System.out.println(a.getNome()); // B
8      System.out.println(b.getNome()); // B
9      System.out.println();
10
11     // chiamata metodi statici
12     System.out.println(A.getNomeStatico()); // A
13     System.out.println(B.getNomeStatico()); // B
14   }
15 }
```

Classi e metodi `final`

Classi e metodi possono essere dichiarati `final`

- di una classe `final` *non* è possibile definire sottoclassi
- un metodo `final` *non* può essere ridefinito nelle sottoclassi

Esempio

La classe `String` è dichiarata `final`

```
1 public final class String {
2   ...
3 }
```

```
1 public class B extends String { // errore: la classe String
2   ...                           // non puo' essere estesa
3 }
```

- Dichiarare una classe `final` ne limita il possibile riuso

Metodi `final`: esempio

Esempio

```
1 public class A {
2   public final int metodo() {
3     return 10;
4   }
5 }
```

```
1 public class B extends A {
2   public int metodo() { // errore: metodo non puo'
3     return 11;          // essere ridefinito
4   }
5 }
```

Classi `final`: esempio

Esempio

```
1 public final class A {
2   // ...
3 }
```

```
1 public class B extends A { // errore: A non puo' essere estesa
2     // ...
3   }
4 }
```

Classe `Object`

La classe `Object`, supertipo di ogni altra classe, definisce alcuni metodi d'istanza utili

- da ridefinire nelle sottoclassi (se necessario)
- tutti gli oggetti forniscono queste funzionalità

Alcuni di questi metodi permettono di:

- ottenere una descrizione dello stato interno dell'oggetto
- confrontare se l'oggetto corrente è "uguale" ad un altro
- ottenere un codice *hash* dell'oggetto, calcolato dal suo stato interno

Classe `Object`: metodo `toString()`

Il metodo `toString` è utilizzato per ottenere una descrizione dello stato interno dell'oggetto

```
1 public class Object {
2   ...
3   public String toString() { ... }
```

- L'implementazione di default restituisce il nome della classe e il valore hash dell'oggetto (es. `Persona@37aef1b0`)

Esempio

```
1 public class Persona {
2   ...
3   public String toString() {
4       return "Nome: " + nome + "; " +
5              "Indirizzo: " + indirizzo;
6   }
```

Classe `Object`: metodo `equals(obj)`

Il metodo `equals` confronta se l'oggetto corrente è "uguale" ad un altro oggetto passato come parametro

- confrontando il contenuto delle variabili d'istanza opportune

```
public class Object {
  ...
  public boolean equals(Object obj) {
      return (this == obj);
  }
```

- L'implementazione di default controlla i riferimenti

Esempio

- La classe `String` ridefinisce il metodo per controllare il contenuto delle due stringhe

Metodo equals(obj): esempio

```java
public class Persona {
  ...
  public boolean equals(Object obj) {
    // controllo se l'altro oggetto e' istanza di Persona
    if (!(obj instanceof Persona))
        return false;

    // controllo se le variabili d'istanza sono uguali
    Persona p = (Persona)obj;
    return this.nome.equals(p.nome) &&
           this.indirizzo.equals(p.indirizzo);
  }
```

Esempio

```java
Persona p1 = new Persona("Pippo", "Pisa");
Persona p2 = new Persona("Pippo", "Pisa");
System.out.println( p1 == p2 );      //false
System.out.println( p1.equals(p2) ); //true
```

Polimorfismo

Sfruttando l'ereditarietà si possono realizzare strutture dati polimorfe

Esempio
La classe `ArrayList`[1] rappresenta una lista di oggetti di dimensione variabile

- la dimensione cambia automaticamente quando gli elementi vengono aggiunti/rimossi
- si accede agli elementi tramite un indice

- La classe `ArrayList` può contenere elementi di tipo `Object`

[1] La classe `ArrayList` è analoga alla `Vector`.

Esempio: classe `ArrayList` (1)

Alcuni metodi forniti dalla classe `ArrayList`

```
// aggiunge un oggetto in fondo alla lista
boolean add(Object o);

// inserisce un oggetto nella posizione specificata
void add(int index, Object o);

// rimuove l'elemento nella posizone specificata
Object remove(int index);

// restituisce la lunghezza della lista
int size();

// restituisce l'elemento alla posizione specificata
Object get(int index);
```

- Non è possibile utilizzare direttamente i tipi primitivi

Esempio: classe `ArrayList` (2)

Esempio

```
1 ArrayList l = new ArrayList();
2 l.add("rosso");
3 l.add("blu");
4 l.add(0, "giallo");
5 l.remove(1);
6
7 System.out.println(l.size()); // 2
8 System.out.println(l.get(0)); // giallo
9 System.out.println(l.get(1)); // blu
```

Sommario

Classi astratte

Una classe astratta può contenere metodi non implementati (metodi astratti)

- per i metodi astratti sono forniti solo il nome e la firma, ma nessuna implementazione
- l'implementazione dei metodi astratti viene fornita dalle sottoclassi concrete (non astratte), ridefinendo i metodo astratti (overriding)

- Per dichiarare classe e metodi astratti si usa la keyword `abstract`

```
abstract class <NomeClasse> {
  // definizione di campi e metodi
  ...

  // definizione di metodo astratto
  [<visibilità>] abstract <tipo> <metodo> (<parametri>);
  ...
}
```

Classi astratte

Vincoli

- una classe che contiene metodi astratti (anche ereditati) deve essere dichiarata anch'essa astratta
- una classe astratta non può essere istanziata
  - può mancare l'implementazione di alcuni metodi

Una classe astratta è tipicamente usata per fattorizzare il codice, ed evitarne la duplicazione

- la classe astratta fornisce solo una parte dell'implementazione
- con i metodi astratti si lascia la libertà alle sottoclassi di specificare alcuni comportamenti

Classi astratte: esempio (1)

La classe Shape fornisce un metodo `area()` per calcolare l'area della figura

- il metodo può essere implementato ragionevolmente solo nelle sottoclassi, ognuna delle quali rappresenta un tipo preciso di figura

Implementazione attuale

```
1 public class Shape {
2   private Point position;
3   private boolean visible;
4   ...
5
6   public double area() {
7     return 0;
8   }
9   ...
```

- Rendiamo astratti la classe Shape e il metodo `area()`

Classi astratte: esempio (2)

```
1  public abstract class Shape {
2    private Point position;
3    private boolean visible;
4
5    public Shape(Point p) {
6      position = p;
7      visible = false;
8    }
9
10   public Shape(Point p, boolean vis) {
11     position = p;
12     visible = vis;
13   }
14
15   // metodo astratto
16   public abstract double area();
17
18   ...
19 }
```

- La classe Shape non è più istanziabile

Classi astratte: esempio (3)

Ogni sottoclasse concreta di Shape deve possedere una definizione del metodo `area()`

- il controllo è effettuato dal compilatore

```
1 public class Circle extends Shape {
2   private double radius;
3   ...
4   public double area() {
5     return radius * radius * Math.PI;
6   }
7   ...
```

```
1 public class Rectangle extends Shape {
2   private double width;
3   private double height;
4   ...
5   public double area() {
6     return height * width;
7   }
8   ...
```

Classi astratte: esempio (4)

Per la sottoclasse `Square` di `Rectangle` non c'è bisogno di ridefinire il metodo `area()`

- viene ereditato quello di `Rectangle`

```
1  public class Square extends Rectangle {
2
3    public Square(Point position, double edge) {
4      super(position, edge, edge);
5    }
6
7    void setHeight(double v) { resize(v,v); }
8
9    void setWidth(double v)  { resize(v,v); }
10
11   void resize(double newWidth, double newHeight) {
12     if (newWidth == newHeight)
13       super.resize(newWidth, newHeight);
14   }
15 }
```

Interfacce

Le interfacce sono classi astratte particolari, che contengono soltanto metodi astratti

- non sono consentite variabili (né d'istanza, né statiche)

```
interface <NomeInterfaccia>
          [ extends <AltraInterfaccia> ] {

  // dichiarazione di metodo
  <tipo> <metodo> (<parametri>);
  ...
}
```

- il livello di accesso di tutti i membri di una interfaccia è sempre `public`, anche se non specificato
- non serve la keyword `abstract`
- similmente alle classi, si possono avere delle gerarchie tra le interfacce
  - una interfaccia può estenderne un'altra

Interfacce

Java non permette ereditarietà multipla, nella quale una classe può essere definita come sottoclasse di più di un'altra classe

- ereditare campi e metodi da più classi può causare ambiguità semantiche *(problema del diamante)*

In Java ogni classe (eccetto `Object`) eredita da una e una sola altra classe, ma può implementare un qualsiasi numero di interfacce *(mix-in inheritance)*

- dalle interfacce si ereditano soltanto metodi astratti
- il problema dell'ambiguità è evitato

- Una interfaccia "descrive" una funzionalità, che è supportata da tutte le classi che implementano l'interfaccia

Interfacce e classi

Per indicare che una classe implementa una o più interfacce si usa la keyword `implements` nella dichiarazione

```
class <NomeClasse>
      [ extends <AltraClasse> ]
      implements <Interfaccia1>, ..., <Interfaccia1>  {

  // definizione dei membri della classe
  ...
}
```

- Come per le classi, anche le interfacce definiscono tipi *(tipi riferimento)*
  - le relazioni di sottotipo/supertipo sono estese anche alle interfacce
  - le proprietà e operazioni sui tipi viste per le classi sono valide anche per le interfacce

Interfaccia `Comparable`

L'interfaccia `Comparable` è utilizzata per confrontare oggetti di una classe

- deve essere definito un ordinamento totale tra gli oggetti

```
public interface Comparable {
    public int compareTo(Object obj);
}
```

- Il metodo `compareTo` confronta l'oggetto corrente con quello `obj` passato come parametro, restituendo:
  - un valore < 0, se l'oggetto corrente è minore di `obj`
  - il valore 0, se l'oggetto corrente è uguale a `obj`
  - un valore > 0, se l'oggetto corrente è maggiore di `obj`
- Varie proprietà andrebbero garantite (es. transitività)

Interfaccia Comparable: esempio (1)

La classe String implementa l'interfaccia Comparable, confrontando le stringhe secondo l'ordinamento lessicografico

Esempio

```
1 String s1 = "aba";
2 String s2 = "aac";
3 String s3 = "abacc";
4 String s4 = s3.substring(0,3);
5
6 Comparable c1 = s1;
7 System.out.println(c1.compareTo(s2)); // > 0
8 System.out.println(c1.compareTo(s3)); // < 0
9 System.out.println(c1.compareTo(s4)); // 0
```

Interfaccia `Comparable`: esempio (2)

Classe `Pair`, per rappresentare coppie di interi $(n1, n2)$

- ancora ordinamento lessicografico

Esempio

```
1  public class Pair implements Comparable {
2    private int n1;
3    private int n2;
4
5    public Pair(int n1, int n2) {
6      this.n1 = n1;
7      this.n2 = n2;
8    }
9
10   ...
```

Interfaccia Comparable: esempio (3)

Metodo compareTo per la classe Pair

```
1   ...
2   public int compareTo(Object obj) {
3     Pair other = (Pair)obj;
4
5     if (n1 == other.n1 && n2 == other.n2)
6       return 0;
7
8     if (n1 < other.n1 || (n1 == other.n1 && n2 < other.n2))
9       return -1;
10    else
11      return +1;
12  }
13 }
```

Interfaccia Comparable: esempio (4)

Esempio

```
1 Pair p1 = new Pair(4, 4);
2 Pair p2 = new Pair(4, 7);
3 Pair p3 = new Pair(2, 9);
4 Pair p4 = new Pair(4, 4);
5
6 Comparable c1 = p1;
7 System.out.println(c1.compareTo(p2)); // -1
8 System.out.println(c1.compareTo(p3)); // 1
9 System.out.println(c1.compareTo(p4)); // 0
```

Sommario

Risoluzione dei nomi

Il linguaggio Java ha scoping statico, quindi tutti i *nomi* (identificatori) possono essere risolti a tempo di compilazione

- ma attenzione ai metodi overridden

Java fornisce due modi per riferire campi/metodi:

- usando un nome non qualificato

```
<nome>                  // variabile
<nome> (<parametri>) // metodo
```

- usando un nome qualificato
  - è specificato l'oggetto o classe in cui cercarlo

```
<espressione> . <nome>                  // variabile
<espressione> . <nome> (<parametri>) // metodo

<NomeClasse> . <nome>                  // variabile
<NomeClasse> . <nome> (<parametri>)  // metodo
```

Risoluzione dei nomi e gerarchia di tipi

In generale, la risoluzione di un nome (non locale ad un metodo) avviene eseguendo i seguenti passi:

1. si determina l'oggetto o la classe da cui iniziare la ricerca
   - dipende dal tipo di espressione (qualificato/non qualificato) e da dove essa appare (metodo statico/d'istanza, ecc.)
2. se è una *classe*, ed un membro della classe ha il nome cercato
   - se il membro è statico, *ok*
   - se il membro è d'istanza, *errore*
3. se è un *oggetto* di una classe `C` *(tipo apparente)*,
   - se un membro d'istanza o statico della classe `C` ha il nome cercato, *ok*
4. se il nome non è definito nella classe di partenza, si risale la gerarchia delle classi, considerando una classe alla volta (in ordine) fino alla radice `Object`

Risoluzione dei nomi non qualificati

Per un nome non qualificato

```
<nome>                  // variabile
<nome> (<parametri>) // metodo
```

la ricerca inizia da:

- l'oggetto corrente, se siamo in un metodo d'istanza
- la classe corrente, se siamo in un metodo statico

Risoluzione dei nomi qualificati

Per un nome qualificato

```
<espressione> . <nome>                  // variabile
<espressione> . <nome> (<parametri>) // metodo
```

la ricerca inizia dall'oggetto ottenuto dalla valutazione di `<espressione>`

Per un nome qualificato

```
<NomeClasse> . <nome>                   // variabile
<NomeClasse> . <nome> (<parametri>)   // metodo
```

la ricerca inizia dalla classe specificata

Risoluzione dei nomi qualificati: esempio

```java
public class A {
  int k = 10;

  int getK() {
    return k;
  }
}
```

```java
public class B extends A {
  static int k = 20;
}
```

```java
public class Test {
  public static void main(String[] args) {
    B b = new B();
    A a = b;
    System.out.println(B.k); // 20: statico B
    System.out.println(b.k); // 20: statico B
    System.out.println(a.k); // 10: d'istanza A

    System.out.println(a.getK()); // 10: A
    System.out.println(b.getK()); // 10: A
  }
}
```

Sommario

Tipi primitivi e conversioni

- **`byte`**: 8-bit signed two's complement integer
  values: -128 . . . 127
- **`short`**: 16-bit signed two's complement integer
  values: -32768 . . . 32767
- **`int`**: 32-bit signed two's complement integer
  values: -2147483648 . . . 2147483647
- **`long`**: 64-bit signed two's complement integer
- **`float/double`**: single/double-precision floating point number
- **`boolean`**: two possible values `true`/`false`
- **`char`**: 16-bit Unicode character (16-bit unsigned integer)
  values: `'\u0000'` (or 0) . . . `'\uffff'` (or 65535)

- Per i tipi primitivi numerici, i valori di un tipo possono essere convertiti in un altro tipo (conversioni implicite/esplicite)

Conversioni

Per i tipi primitivi numerici (non `boolean`), un valore di un tipo può essere sempre convertito in un altro tipo:

- implicitamente, se non si ha perdita di precisione
- esplicitamente, con una operazione di cast

Cast

Il cast esplicito di un valore usa la seguente sintassi:

```
(<tipo>) <espressione>
```

Conversioni: esempio (1)

Esempio

```
1 int i = 120;
2 long l = i;  // cast implicito
3 short s = i; // non consentito implicitamente
```

Esempio

```
1 int i = 120;
2 short s = (short)i; // cast esplicito
```

Esempio

```
1 char c1 = 'Z';
2 int i = c1;         // cast implicito
3 char c2 = (char)i; // cast esplicito
```

- I cast fra tipi primitivi numerici non falliscono mai

Classi *wrapper* (1)

Nella libraria di Java sono definite, per ogni tipo primitivo, delle classi wrapper per incapsulare un valore primitivo in un oggetto

- tipo `byte`: classe `Byte`
- tipo `short`: classe `Short`
- tipo `int`: classe `Integer`
- tipo `long`: classe `Long`
- tipo `float`/`double`: classi `Float`/`Double`
- tipo `boolean`: classe `Boolean`
- tipo `char`: classe `Character`

- Gli oggetti *non* sono modificabili
- Il linguaggio converte automaticamente tra tipi primitivi e loro classi wrapper *(auto-boxing/unboxing)*
- Queste classi forniscono anche metodi per convertire i valori da/in stringhe

Classi *wrapper* (2)

Ogni classe wrapper fornisce:

- un costruttore con un unico parametro del tipo primitivo corrispondente
- un metodo per ottenere il valore contenuto nell'oggetto

```
<tipo primitivo> nometipoValue()
```

Esempio

```
1 Integer io = null;
2 io = new Integer(51);
3 int v = io.intValue();
```

Esempio

```
1 Integer d;
2 d = 4;       // auto boxing
3 int k1 = d.intValue();
4 int k2 = d; // auto unboxing
```

Esempio

```
1 Integer io = new Integer(51);
2 String s = io.toString();
3 System.out.println(s); // 51
4
5 int k = Integer.parseInt("21");
```

Sommario

Array

Un array rappresenta una sequenza finita di variabili, accedibili tramite un indice intero

- sono composti da elementi di uno stesso tipo
- gli indici di un array di lunghezza n sono $0, 1, \ldots, n-1$

Gli array definiscono tipi parametrici

- a seconda del tipo degli elementi contenuti, si ha un tipo diverso

```
int[]    // array di int
Point[]  // array di (riferimenti a) oggetti di tipo Point
```

- Gli array sono *oggetti*, quindi sono allocati sullo heap
- Se il tipo degli elementi è una Classe, l'array contiene riferimenti ad oggetti della classe

Creazione di un array

Un array viene creato specificando il tipo degli elementi e la lunghezza

```
new <tipo>[<lunghezza>];
```

Esempio

```
1  int[] list;
2  list = new int[3];
3  Point[] ps = new Point[5];
```

- Gli elementi sono automaticamente inizializzati al loro valore nullo (dipende dal tipo)

Accesso agli elementi di un array

Per accedere ad un elemento di un array si deve specificare l'indice

```
<array> [ <indice> ]
```

Esempio

```
1  int[] list = new int[3];
2  list[0] = 10;
3  list[1] = 20;
4  list[2] = 30;
5  int x = list[0];
```

- Non è possibile accedere oltre i limiti dell'array
  - tutti gli accessi sono controllati a run-time (`IndexOutOfBoundsException`)

Creazione e inizializzazione di un array

È possibile assegnare un valore agli elementi dell'array contestualmente alla creazione dell'array

```
new <tipo>[] { <elem1>, <elem2>, ..., <elemk> };
```

- La lunghezza dell'array viene automaticamente impostata al numero degli elementi specificati

Esempio

```
1  // array di 4 elementi
2  int[] list = new int[] { 10, 20, 30, 40 };
```

Esempio

```
1  Point p1 = new Point(9, 4);
2
3  // array di 3 elementi
4  Point[] ps = new Point[] { p1, new Point(9,4), p1 };
```

Sommario I

Sommario II

- Classi wrapper

5 Array

Sommario

Operatore '.'

L'operatore '.', per riferire i membri di una classe, è associativo a sinistra

- la valutazione avviene da sinistra verso destra

Una espressione

```
<exp1> . <exp2> . <exp3> .  <exp5>  . <exp6>
```

viene interpretata come

```
(((<exp1> . <exp2>) . <exp3>) . <exp5>) . <exp6>
```

Esempio

```
1  System.out.println("Hello World!");
2  (System.out).println("Hello World!");
```

Chiamata del metodo `println` sull'oggetto riferito dalla variabile statica `out` della classe `System`

Inizializzazione dei campi

I campi di una classe (variabili statiche e d'istanza) vengono inizializzate automaticamente al loro valore nullo

- prima dell'esecuzione del costruttore

Il valore nullo di un campo dipende dal tipo con cui è dichiarato:

- tipi primimitivi numerici `int`, `float`, ecc.: 0 *(zero)*
- tipo `boolean`: `false`
- tipi riferimento (classi e interfacce): `null`

- È possibile inizializzare un campo direttamente dove è dichiarato

```
<tipo> <nomeCampo> = <espressione>;
```

Inizializzazione dei campi: esempio

```
1 public abstract class Shape {
2   private Point position = new Point(0, 0);
3   private boolean visible = true;
4
5   ...
6 }
```

Costanti

Un campo di una classe può essere dichiarato costante usando la keyword `final`

- deve essere inizializzato esplicitamente
  - direttamente al momento della dichiarazione, o nel costruttore
- una volta inizializzato non può più essere modificato

- Le costanti sono, di solito, dichiarate come campi statici
  - sono permesse dichiarazioni di costanti anche nelle interfacce

Esempio

```
1 public class CardinalPoints {
2   public static final String NORTH = "North";
3   public static final String SOUTH = "South";
4   public static final String EAST = "East";
5   public static final String WEST = "West";
6 }
```