Linguaggi di Programmazione I – Lezione 11

Prof. Marcello Sette
mailto://marcello.sette@gmail.com
http://sette.dnsalias.org

13 maggio 2008

Ereditarietà **3**
Specializzazione (1) . . . 4
Specializzazione (2) . . . 5
Specializzazione (3) . . . 6
Polimorfismo . . . 7
Collezioni eterogenee . . . 8
Argomenti polimorf. . . . 9
Oper. `instanceof` . . . 10
Casting (1). . . . 11
Casting (2). . . . 12
Casting (3). . . . 13

Altre relazioni tra classi **14**
Composizioni . . . 15
Aggregazioni . . . 16
Associazioni . . . 17
Molteplicità . . . 18

Overloading e overriding **19**
Overl. di metodi . . . 20
Overl. di costruttori . . . 21
Overr. di metodi (1) . . . 22
Overr. di metodi (2) . . . 23
Overr. di metodi (3) . . . 24

Costruzione di oggetti **25**
`super` . . . 26
`super`-costruttore . . . 27
Costr. di oggetti . . . 28
Esempio (1.a) . . . 29
Esempio (1.b) . . . 30
Esempio (2.a) . . . 31
Esempio (2.b) . . . 32

La classe `Object` **33**
La classe `Object` . . . 34

Il metodo `equals` . . . 35
- Esempio (1) . . . 36
- Esempio (2) . . . 37
Il metodo `toString` . . . 38

Le classi "wrapper" **39**
Definizione . . . 40
Uso . . . 41

Esercizi **42**
Esercizi . . . 42

Panoramica della lezione

Ereditarietà

Altre relazioni tra classi

Overloading e overriding

Costruzione di oggetti

La classe `Object`

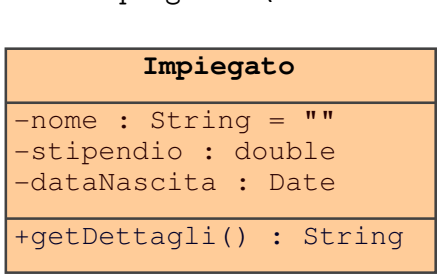
Le classi "wrapper"

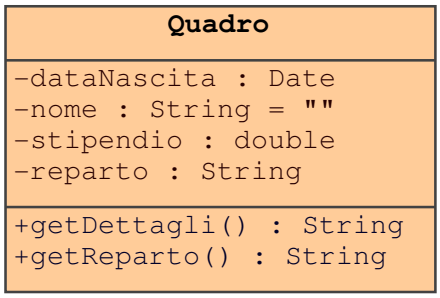
Esercizi

Ereditarietà

Specializzazione (1)

Le classi `Impiegato` e `Quadro`:



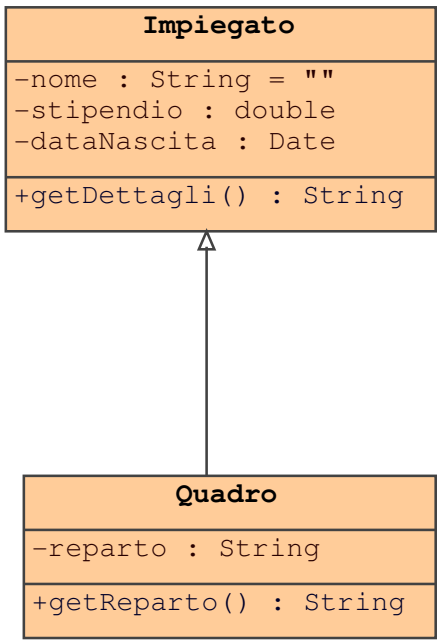



```
public class Impiegato {
  private String nome = "";
  private double stipendio;
  private Date dataNascita;

  public String getDettagli() {
    ...
  }
}
```

```
public class Quadro {
  private String nome = "";
  private double stipendio;
  private Date dataNascita;
  private String reparto;

  public String getDettagli() {...}
  public String getReparto() {
    return reparto;
  }
}
```

Specializzazione (2)

Ridefinizione:




```
public class Impiegato {
  private String nome = "";
  private double stipendio;
  private Date dataNascita;

  public String getDettagli() {
    ...
  }
}

public class Quadro
  extends Impiegato {
  private String reparto;
  public String getReparto() {
    return reparto;
  }
}
```


Specializzazione (3)

- Una sottoclasse eredita tutti i metodi e le variabili della superclasse.
- Una sottoclasse NON eredita i costruttori della superclasse.
- I due modi, esclusivi uno con l'altro, per includere i costruttori in una classe sono:
  - usare il costruttore di default (senza parametri);
  - scrivere uno o più costruttori espliciti.

Polimorfismo

- Un oggetto ha solo una forma.
- Polimorfismo è l'abilità di riferirsi con la medesima variabile ad oggetti con forme diverse (credendo, però, di riferirsi ad un oggetto di una particolare forma ed ignorando, cioè, la reale forma dell'oggetto).
- Poiché si può dire sia:

```
Impiegato impiegato = new Impiegato();
```

sia:

```
Impiegato impiegato = new Quadro();
```

la stessa variabile `impiegato` può essere usata per riferirsi ai due oggetti distinti; l'utilizzatore della variabile in entrambi i casi crederà di riferirsi ad una istanza di `Impiegato`.

- Non sarà, pertanto, mai legale scrivere:

```
String reparto = impiegato.getReparto();
```

anche se `impiegato` fosse in realtà un riferimento ad una istanza di `Quadro`.

Collezioni eterogenee

- Collezioni *omogenee* di oggetti della stessa classe:

```
MiaData[] date = new MiaData[2];
date[0] = new MiaData(5, 5, 2006);
data[1] = new MiaData(25,12,2006);
```

- Collezioni *eterogenee* di oggetti di classi diverse:

```
Impiegato[] staff = new Impiegato[100];
staff[0] = new Impiegato();
staff[1] = new Impiegato();
staff[2] = new Quadro();
```


Argomenti polimorfici

- Si possono costruire metodi che accettino come parametro un riferimento "generico" ed operare in modo automatico su un più vasto insieme di oggetti.
- In questo caso è il metodo ad ignorare la reale natura (forma) dell'oggetto che gli viene specificato come parametro attuale.
- Per esempio, poiché un `Quadro` è un `Impiegato`:

```
// Nella classe A
public double getIrpef(Impiegato i) {
  ...
}

// In un'altra classe:
A a = new A();
Quadro q = new Quadro();

double irpef = a.getIrpef(q)
```

L'operatore `instanceof`

- Poiché è lecito scambiarsi oggetti usando riferimenti ai loro antenati (nella gerarchia ad albero), potrebbe essere a volte necessario sapere esattamente la forma dell'oggetto con cui si ha a che fare.
- Per esempio, supponiamo che vi sia una gerarchia di classi:

```
public class Impiegato extends Object // ridondante
public class Quadro extends Impiegato
public class Segretario extends Impiegato
```

Se si riceve un oggetto usando un riferimento ad `Impiegato`, esso potrebbe essere anche realmente un `Quadro` o un `Segretario`. Per saperlo:

```
public void faQualcosa(Impiegato i) {
  if (i instanceof Quadro) {
    // elabora un Quadro
  } else if (i instanceof Segretario) {
    // elabora un Segretario
  } else {
    // elabora un Impiegato qualunque
  }
}
```


Casting (1)

- Nell'eventualità che si riceva un riferimento ad un oggetto e che (usando `instanceof`) si sappia che l'oggetto è realmente di una sottoclasse, si dovrebbe comunque usare quell'oggetto come se fosse della superclasse.
- La formalizzazione di polimorfismo rende invisibili tutte le specializzazioni proprie di quell'oggetto.
- Per rendere visibili tutte le caratteristiche dell'oggetto occorre fare un "cast" esplicito:

```
public void faQualcosa(Impiegato i) {
  if (i instanceof Quadro) {
    Quadro q = (Quadro) i;
    System.out.println(
      "Questo e' il coordinatore del reparto " +
      q.getReparto());
  }
  ...
}
```

- Se non ci fosse il cast, il metodo `getReparto` sarebbe stato inaccessibile al compilatore.

Casting (2)

Ogni tentativo di effettuare un cast di riferimenti ad oggetti è soggetto a regole precise. A tale scopo, siano `A`, `B` e `C` tre tipi:

- Gli "upcast" sono sempre permessi e, infatti, non richiedono l'operatore di cast. Essi sono realizzati con una semplice assegnazione. Il codice:

```
C c;
...
B b = c;
```

è corretto a patto che `C` sia sottoclasse di `B`, indipendentemente dall'oggetto a cui si riferisce `c`.

Casting (3)

- Per i "downcast" il compilatore controlla che l'operazione sia almeno possibile: la classe del riferimento di destinazione deve essere una sottoclasse del riferimento di origine. Il codice:

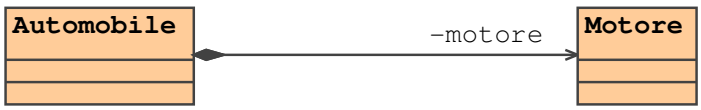
```
B b;
...
C c = (C) b;
```

  è corretto in compilazione se `C` è sottoclasse di `B`.
- Una volta che il compilatore abbia ammesso l'operazione, essa sarà infine controllata a runtime, per verificare che il tipo dell'oggetto riferito sia compatibile con il tipo del riferimento di destinazione. Se, per esempio, si omettesse il controllo con `instanceof` e si realizzasse un downcast con un oggetto che non è realmente del tipo giusto (quello di destinazione, per intenderci), verrà lanciata una eccezione in esecuzione.
- Le regole complete che governano i cast (poiché coinvolgono anche classi astratte ed interfacce) saranno discusse in una lezione successiva.

Altre relazioni tra classi 14 / 42

Composizioni

- La composizione implica una relazione a vita.
- Se il tutto è creato, anche le parti sono create.
- Quando il tutto muore, anche le parti muoiono.
- Il costruttore per il tutto deve costruire anche le parti.
- Esempio:




```
public class Automobile {
  private Motore motore;
  // altri attributi

  public Automobile (int cilindrata, int numeroCilindri,
                     String marca, String modello) {
    motore = new Motore (cilindrata, numeroCilindri);
    // altro codice
  }

  // altri metodi
}
```

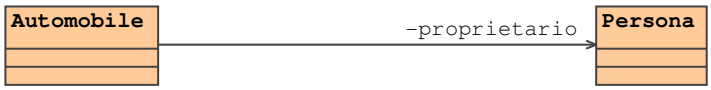
Aggregazioni

- Nella aggregazione le parti sono create altrove. Esse sono inglobate nell'aggregante nel momento in cui esso viene costruito.
- Esiste un loro riferimento anche al di fuori dell'aggregante: in questo modo esse non cessano (forse) di esistere dopo la morte di esso. Esempio:


```
public class Automobile {
  private Motore motore;
  // altri attributi

  public Automobile(Motore motore, String marca, String modello) {
    this.motore = motore;
    // altro codice
  }

  // altri metodi
}

// altrove nel codice
Motore mioMotore = new Motore (1400, 3);
Automobile miaAuto = new Automobile (mioMotore, "Seat", "Ibiza");
```

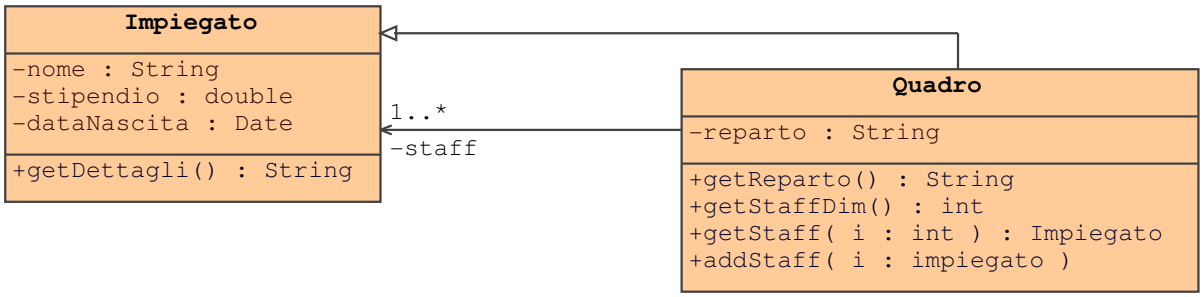

Associazioni

- Nella semplice associazione le vite degli oggetti, sia pure correlate, esistono in modo totalmente indipendente.
- Esempio




```
public class Automobile {
  private Persona proprietario;
  // altri attributi

  public void setProprietario (Persona proprietario) {
    this.proprietario = proprietario;
  }

  // altri metodi
}

// altrove nel codice
Automobile miaAuto = new Automobile ("Seat", "Ibiza");
Persona io = new Persona ("Marcello", "Sette");
miaAuto.setProprietario(io);
```

Molteplicità




```
public class Quadro extends Impiegato {
  private String reparto = "";
  private Impiegato[] staff = new Impiegato[20];
  private int staffDim = 0;

  public void getReparto() { return reparto }
  public int getStaffDim() { return staffDim }
  public Impiegato getStaff(i:int) { return staff[i] }
  public void addStaff(i:Impiegato) { staff[staffDim++] = i }
}
```


Overloading e overriding 19 / 42

Overloading di metodi

- Metodi **con lo stesso nome**, **nella stessa classe**, che svolgono lo stesso compito con diversi argomenti.
- Per esempio:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

- La lista degli argomenti DEVE essere diversa, perché essa sola è usata per distinguere i metodi.
- Il tipo di ritorno PUÒ essere diverso ma non è usato per distinguere i metodi (cioè, metodi con uguale nome e lista di parametri, ma diverso tipo di ritorno sono indistinguibili ed il compilatore segnale l'errore).

Overloading di costruttori

- Stesse regole dei metodi.
- Il riferimento `this` può essere usato, NELLA PRIMA LINEA di un costruttore, per invocare un altro costruttore.

```
public class Impiegato {
  private static final double STIPENDIO_BASE = 15000.00;
  private String nome; private double stipendio;
  private Date dataNascita;

  public Impiegato(String nome, double stip, Date nasc) {
    this.nome=nome; stipendio=stip; dataNascita=nasc;
  }
  public Impiegato(String nome, double stip) {
    this(nome, stip, null);
  }
  public Impiegato(String nome, Date nasc) {
    this(nome, STIPENDIO_BASE, nasc);
  }
  public Impiegato(String nome) {
    this(nome, STIPENDIO_BASE);
  }
  // Altro codice
}
```


Overriding di metodi (1)

Una sottoclasse può sovrapporre un metodo ereditato e visibile, a patto che, rispetto al metodo della superclasse, il nuovo metodo abbia (a) lo stesso nome, tipo di ritorno e lista di argomenti; (b) visibilità non inferiore [vedremo poi].

```
public class Impiegato {
  protected String nome; protected double stipendio;
  protected Date dataNascita;

  public String getDettagli() {
    return "Nome: " + nome + "\n" +
           "Stipendio: + stipendio;
  }
}

public class Quadro extends Impiegato {
  protected String reparto;

  public String getDettagli() {
    return "Nome: " + nome + "\n" +
           "Stipendio: + stipendio + "\n" +
           "Reparto: + reparto;
  }
}
```

Overriding di metodi (2)

- In riferimento all'esempio precedente, è chiaro quali metodi sono invocati in:

```
Impiegato i = new Impiegato();
Quadro q = new Quadro();

System.out.println(i.getDettagli());
System.out.println(q.getDettagli());
```

- Ma quale metodo è invocato qui?

```
Impiegato i = new Quadro();

System.out.println(i.getDettagli());
```

- Si ottiene il comportamento del tipo che assumerà la variabile in esecuzione e non il comportamento del tipo che la variabile ha in compilazione (invocazione virtuale di metodo).
- L'invocazione virtuale dei metodi è la massima rivelazione del polimorfismo.

Overriding di metodi (3)

Riguardo al vincolo sulla non inferiore visibilità, il codice seguente non viene compilato (a dispetto del fatto che la vera risoluzione di tipo avviene in esecuzione):

```
public class Padre {
  public void faQualcosa() { }
}

public class Figlio extends Padre {
  private void faQualcosa() { }
}

public class UsaEntrambi {
  public void faAltro() {
    Padre p1 = new Padre();
    Padre p2 = new Figlio();
    p1.faQualcosa();
    p2.faQualcosa();
  }
}
```

La parola chiave `super`

- `super` è usata in una classe per riferirsi alla superclasse.
- `super` è usata per riferirsi ai membri della superclasse (attributi e metodi) con la dot notation.
- I membri della superclasse accessibili con `super` sono anche quelli che la superclasse ha ereditato e non solo quelli esplicitamente definiti in essa.
- NON si può usare `super.super.membro` per accedere a membri di classi di livello superiore alla prima superclasse.

```
public class Impiegato {
  private String nome;
  private double stipendio;

  public String getDettagli() {
    return "Nome: " + nome +
    "\nStipendio: " + stipendio;
  }
}

public class Quadro
  extends Impiegato {
  private String reparto;

  public String getDettagli() {
    return
      super.getDettagli() +
      "\nReparto: " + reparto;
  }
}
```


Invocazione di un `super`-costruttore

- Si può invocare uno dei costruttori della superclasse usando `super(...)` come PRIMA linea nel proprio costruttore, con analoga sintassi di quella del proprio costruttore (e.g. `this(...)`).
- Se, come prima linea nel proprio costruttore, non c'è nè `this`, nè `super`, il compilatore pone una chiamata implicita a `super()`, cioè al costruttore di default della superclasse.
  - In questo caso, se non esiste il costruttore di default nella superclasse, il compilatore genera un errore.

```
public class Quadro
  extends Impiegato {
  private String reparto;

  public Quadro(String nome,
                double stipendio,
                String reparto) {
    super(nome, stipendio);
    this.reparto = reparto;
  }
  public Quadro(String nome,
                String reparto) {
    super(nome);
    this.reparto = reparto;
  }
  public Quadro(String reparto) {
    this.reparto = reparto;
    // errore: assenza del
    // costruttore Impiegato()
    // nella superclasse
  }
}
```

Costruzione ed inizializzazione di oggetti

Riprendiamo

- Viene allocata nello heap la memoria per il nuovo oggetto e sono inizializzate le variabili di istanza ai valori impliciti di default.
- Per ogni successiva chiamata ad un costruttore, cominciando dal costruttore individuato per primo, viene eseguita la sequenza di passi:
  1. Assegna i parametri del costruttore.
  2. Se è presente, come prima riga, una chiamata esplicita a `this(...)`, richiama ricorsivamente il nuovo costruttore e poi passa al punto 5.
  3. Richiama l'implicito o esplicito `super(...)`-costruttore, eccetto per `Object`, poiché `Object` non ha superclasse.
  4. Esegue ogni inizializzazione esplicita delle variabili di istanza.
  5. Esegue il corpo del costruttore corrente.

Esempio (1.a)

Esecuzione di `new Quadro("M 7", "vendite")`.

```
public class Impiegato {
  private String nome;
  private double stipendio=15000.00;
  private Date dataNascita;

  public Impiegato(String n, Date d) {
    nome = n;
    dataNascita = d;
  }
  public Impiegato(String n) {
    this(n, null);
  }
}

public class Quadro extends Impiegato {
  private String reparto;

  public Quadro(String n, String r) {
    super(n); // e se manca sta riga?
    reparto = r;
  }
}
```

Esempio (1.b)

0 Inizializzazione base (`new ...`).

- 0.1 Allocazione memoria per un oggetto `Quadro`.
- 0.2 Inizializzazione di `reparto` al valore di default `null`.

1 Esecuzione del costruttore: `Quadro("M 7", "vendite")`.

- 1.1 Inizializzazione dei parametri del costruttore: `n="M 7"`, `r="vendite"`.
- 1.2 Non c'è chiamata a `this(...)`.
- 1.3 Invocazione di `super("M 7")` come istanza di `Impiegato(String n)`.
  - 1.3.1 Inizializzazione del parametro del costruttore: `n="M 7"`.
  - 1.3.2 Esecuzione di `this("M 7", null)` come istanza di `Impiegato(String n, Date d)`.
    - 1.3.2.1 Inizializzazione dei parametri del costruttore: `n="M 7"`, `d=null`.
    - 1.3.2.2 Nessuna chiamata esplicita a `this(...)`.
    - 1.3.2.3 Nessuna chiamata esplicita a `super(...)`: viene eseguito implicitamente `super()` come istanza di `Object()` (la classe `Object` è la radice di tutti gli oggetti).
      - 1.3.2.3.1 Nessuna inizializzazione di parametri.
      - 1.3.2.3.2 Nessuna chiamata a `this(...)`.
      - 1.3.2.3.3 Nessuna chiamata a `super(...)` (`Object` è la radice).
      - 1.3.2.3.4 Nessuna inizializzazione esplicita.
      - 1.3.2.3.5 Nessun corpo da eseguire.
    - 1.3.2.4 Inizializzazione esplicita della variabile: `stipendio=15000.00`.
    - 1.3.2.5 Esecuzione del corpo del costruttore: `nome="M 7"`, `dataNascita=null`.
  - 1.3.3 Saltato.
  - 1.3.4 Saltato.
  - 1.3.5 Nessun corpo in `Impiegato(String n)`.
- 1.4 Nessuna inizializzazione esplicita per `Quadro`.
- 1.5 Esecuzione di `reparto="Vendite"`.

Esempio (2.a)

```
public class Impiegato {
  private String nome;
  private double
    stipendio=15000.00;
  private Date dataNascita;
  private String sommario;

  public Impiegato(String n,
                   Date d) {
    nome = n; dataNascita = d;
    sommario = getDettagli();
  }
  public Impiegato(String n) {
    this(n, null);
  }

  public String getDettagli(){
    return "Nome: " + nome +
      "\nStipendio: " + stipendio
      + "\nData di nascita: " +
      dataNascita;
  }
}
```

```
public class Quadro
  extends Impiegato {
  private String reparto;

  public Quadro(String n,
                String r) {
    super(n);
    reparto = r;
  }

  public String getDettagli(){
    return super.getDettagli() +
      "\nReparto: " + reparto;
  }
}
```

Esempio (2.b)

Quello che succede, nella generazione di una istanza di `Quadro` mediante `new Quadro("M 7", "Vendite")`, è che:

- il metodo `getDettagli()` viene sovrapposto nella classe `Quadro`;
- viene invocato `getDettagli()` nel costruttore della superclasse `Impiegato`;
- a runtime viene scelto il metodo di `Quadro` (invocazione virtuale di metodi);
- la variabile `reparto` non è stata, però, ancora esplicitamente inizializzata: piccolo problema, perché il programma viene comunque eseguito, ma viene aggiunta alla stringa di sommario "`Reparto: null`";
- tutto ciò avrebbe potuto essere catastrofico, nel caso in cui la variabile fosse stata non una variabile di heap, ma una variabile di stack (in questo caso neppure implicitamente inizializzata).

Regola aurea: **Se si invoca un metodo in un costruttore, rendere quel metodo privato!**

In questo modo il metodo non può risultare sovrapposto nella sottoclasse, perché essa pure ereditandolo, non può vederlo!!!

La classe `Object` 33 / 42

La classe `Object`

- In Java la classe `Object` è la radice di tutte le classi.
- Una dichiarazione di classe senza la clausola `extends`, implicitamente usa "`extends Object`". Pertanto

```
public class Impiegato {
  ...
}
```

è equivalente a

```
public class Impiegato extends Object {
  ...
}
```

- Questo ci permette di sovrapporre numerosi metodi della classe `Object`, due dei quali saranno discussi qui di seguito.

Il metodo `equals`

- L'operatore == determina se due riferimenti sono identici (cioè se riferiscono allo stesso, unico, oggetto).
- Il metodo `equals` determina se le due variabili si riferiscono ad oggetti (anche distinti) appartenenti alla stessa classe di equivalenza, rispetto ad una particolare relazione di equivalenza definita dall'utente.
- La realizzazione di `equals` nella classe `Object` fa uso di ==. Quindi le classi di equivalenza di == e di `equals` coincidono in `Object` (relazione di identità).
- Lo scopo del metodo è quello di essere sovrapposto nelle classi sviluppate dagli utenti, in modo da realizzare nuove relazioni di equivalenza.
- Alcune classi di sistema lo fanno già. Per esempio la classe `String` realizza il proprio `equals`, confrontando le stringhe carattere per carattere.
- Viene raccomandato di sovrapporre, insieme con `equals`, anche il metodo `hashCode`. Una decente, anche se povera, realizzazione potrebbe usare l'XOR bit a bit dei codici hash degli oggetti in esame.

- Esempio (1)

```
public class Impiegato {
  private String nome;
  private MiaData nascita;
  private double stipendio;

  public Impiegato(String n, Date d, double s) {
    nome=n; dataNascita=d; stipendio=s;
  }

  public boolean equals(Object o) {
    boolean risultato = false;
    if ((o != null) && (o instanceof Impiegato)) {
      Impiegato i = (Impiegato) o;
      if (nome.equals(i.nome) && nascita.equals(i.nascita)) {
        risultato = true;
      }
    }
    return risultato;
  }

  public int hashCode() {
    return (nome.hashCode() ^ nascita.hashCode());
  }
}
```


- Esempio (2)

```
public class TestEquals {
  public static void main(String[] args) {
    Impiegato i1 = new Impiegato("Eliana",
                                 new MiaData(23, 4, 1964),
                                 25000,00);
    Impiegato i2 = new Impiegato("Marcello",
                                 new MiaData(23, 4, 1964),
                                 25000,00);

    System.out.println("i1 identico ad i2:    " +
                       (i1 == i2));
    System.out.println("i1 equivalente ad i2: " +
                       i1.equals(i2));

    i2 = i1;
    System.out.println("\nPoniamo i2=i1");
    System.out.println("i1 identico ad i2:    " +
                       (i1 == i2));
  }
}
```

Il metodo `toString`

- Restituisce una rappresentazione `String` di un qualunque oggetto.
- Viene usato durante le conversioni automatiche a stringhe. Per esempio:

```
Date now = new Date();
System.out.println(now);
```

è rozzamente equivalente a:

```
Date now = new Date();
System.out.println(now.toString());
```

- Occorre sovrapporre questo metodo quando si vuole fornire una rappresentazione di un oggetto in forma (umanamente) leggibile.
- I tipi primitivi sono rappresentati in forma di `String` usando il metodo statico `toString` della corrispondente classe "wrapper".

Le classi "wrapper" 39 / 42

Definizione

- I tipi primitivi non sono oggetti. Se si vogliono manipolare come oggetti, è possibile avvolgere un SINGOLO valore con una opportuno oggetto, cosiddetto "wrapper". La corrispondenza è questa:

Tipo primitivo	Classe wrapper
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Uso

- Si può costruire un oggetto wrapper, passando il relativo valore al costruttore appropriato. Per esempio:

```
int i = 500;
Integer indice = new Integer(i);
int x = indice.intValue();
```

- Il valore da avvolgere può essere passato anche in una rappresentazione sotto forma di `String`.
- Il valore avvolto può essere estratto usando il metodo opportuno `...Value()`.
- Le classi wrapper sono utili quando si vogliono convertire i tipi primitivi. Per esempio:

```
int x = Integer.valueOf(str).intValue();
int y = Integer.parseInt(str);
```


Esercizi 42 / 42

Esercizi

1. In questo esercizio verranno create due sottoclassi della classe `Conto` nel progetto `banca`: `ContoCorrente` e `LibrettoRisparmio`.
   Si dovrà sovrapporre il metodo `preleva` in `ContoCorrente` ed usare `super` per invocare il costruttore della superclasse.
2. Come ulteriore continuazione del progetto di gestione bancaria, per ciascun cliente della banca in questo esercizio verrà creata una collezione eterogenea di conti (max. 5), cioè lo stesso cliente può essere titolare di conti di diverso tipo.