Lezione 10
Ereditarietà

L'ereditarietà è tra i concetti più importanti della programmazione orientata agli oggetti. Attraverso l'ereditarietà, è possibile raggruppare le classi e gli oggetti secondo un'organizzazione di tipo gerarchico. Questo approccio, come si scoprirà tra poco, facilita la stesura delle applicazioni, deponendo a favore della riusabilità del codice.

10.1 - Primo approccio all'ereditarietà

La parola chiave *extends*, in Java, permette di allacciare rapporti di ereditarietà tra due classi. Avviciniamoci all'argomento mediante un primo esempio pratico. Si prenda in considerazione la seguente definizione della semplice classe *Persona*:

```
class Persona {

  String nome;
  String cognome;

  public void setNome(String nome) {
    this.nome = nome;
  }

  public void setCognome(String cognome) {
    this.cognome = cognome;
  }

  public String getNome() {
    return nome;
  }

  public String getCognome() {
    return cognome;
  }

}
```

Persona non presenta alcuna caratteristica peculiare. Possiede giusto quattro metodi, utili per impostare e recuperare i campi *nome* e *cognome*. Nella rappresentazione usata, queste due proprietà realizzano l'astrazione di un qualsiasi essere umano. Ovviamente, la rappresentazione è volutamente generica. Prendiamo ora in esame la classe *Studente*, che eredita da *Persona*:

```
class Studente extends Persona {

  String matricola;

  public void setMatricola(String matricola) {
    this.matricola = matricola;
  }

  public String getMatricola() {
    return matricola;
```

```
  }

}
```

Prima di entrare nei dettagli di quanto realizzato, si esegua un test dimostrativo:

```
class Test {

  public static void main(String[] args) {
    Studente s = new Studente();
    s.setNome("Mario");
    s.setCognome("Rossi");
    s.setMatricola("09204167");
    System.out.println("Nome: " + s.getNome());
    System.out.println("Cognome: " + s.getCognome());
    System.out.println("Matricola: " + s.getMatricola());
  }

}
```

L'output prodotto é:

```
Nome: Mario
Cognome: Rossi
Matricola: 09204167
```

Cosa c'è di particolare in questo esempio? Per come è stata definita, la classe *Studente* non contiene né il campo *nome*, né il campo *cognome*, né i metodi utili per lavorare con queste due proprietà. A prima vista, sembrerebbe che ciascuno studente venga rappresentato solo ed esclusivamente attraverso la sua matricola. La pratica, però, smentisce queste superficiali osservazioni. Sono stati dati un nome ed un cognome ad un oggetto *Studente*, senza incappare in errori di compilazione o di esecuzione. Ovviamente, ciò avviene in virtù dell'ereditarietà. In particolare, è possibile realizzare quanto visto grazie alla clausola *extends*:

```
class Studente extends Persona {
  // ...
}
```

La classe *Studente* ha ereditato le proprietà ed i metodi della classe *Persona*. Quindi, ogni studente istanziato avrà un campo *matricola*, la cui definizione è data normalmente nel corpo della classe *Studente*, ma avrà anche dei campi *nome* e *cognome*, che derivano da *Persona*.

Giunge il momento si esaminare un po' di utile terminologia. In un caso come quello appena esaminato, si dice che *Persona* è *superclasse* di *Studente*, e che *Studente* è *sottoclasse* di *Persona*. Più in breve, si può dire che *Studente estende Persona* (che poi è il significato letterale della parola chiave *extends*). La classe *Studente deriva da Persona*, cioè gli oggetti di tipo *Studente* ereditano tutte le caratteristiche degli oggetti di tipo *Persona*.

Ecco un secondo test:

```
class Test2 {
```

```
  public static void main(String[] args) {
    Studente s = new Studente();
    s.setNome("Mario");
    s.setCognome("Rossi");
    s.setMatricola("09204167");
    System.out.println("Nome: " + s.getNome());
    System.out.println("Cognome: " + s.getCognome());
    System.out.println("Matricola: " + s.getMatricola());
    System.out.println("toString(): " + s.toString());
  }

}
```

Prima della chiusura del metodo *main()*, è stata aggiunta una nuova istruzione:

```
System.out.println("toString(): " + s.toString());
```

Mandando in esecuzione la classe *Test2*, si ottiene un output analogo al seguente:

```
Nome: Mario
Cognome: Rossi
Matricola: 09204167
toString(): Studente@a1807c
```

Da dove è saltato fuori il metodo *toString()*? La classe *Studente* non definisce nulla di simile. Non si può neanche immediatamente dire che il metodo sia stato ereditato da *Persona*, perché anche qui non esiste alcuna definizione che rispecchi le aspettative. Emerge, così, una caratteristica finora taciuta: in Java, ogni classe che non estende esplicitamente un'altra classe, come *Studente*, eredita direttamente ed implicitamente dalla classe *Object*, definita nella libreria base della piattaforma. In pratica, scrivere

```
public class MiaClasse {
  // ...
}
```

è proprio come digitare

```
public class MiaClasse extends Object {
  // ...
}
```

Quindi, tornando all'esempio di *Persona* e *Studente*, è stata implementata una gerarchia così strutturata:

```
Object
  |
  +- Persona
       |
       +- Studente
```

Studente estende *Persona*, che a sua volta estende *Object*. Il mistero è ora un po' meno fitto:

toString() è un metodo definito da *Object*. *Persona* lo ha ereditato da *Object*, e *Studente* l'ha ereditato da *Persona*. Grazie alla gerarchia messa in atto, il metodo *toString()* è arrivato da *Object* a *Studente*. Sulle caratteristiche della classe *Object* si tornerà più avanti.

Nulla impedisce di derivare ulteriormente una sottoclasse, facendola divenire superclasse di un nuovo elemento. In questo modo è possibile realizzare gerarchie a più livelli. Ad esempio, si riprenda la classe *Studente*, estendendola nella seguente *StudenteIngegneria*:

```
class StudenteIngegneria extends Studente {

  String tipo;

  public void setTipo(String tipo) {
    this.tipo = tipo;
  }

  public String getTipo() {
    return tipo;
  }

}
```

Ecco un veloce test per la verifica:

```
class Test3 {

  public static void main(String[] args) {
    StudenteIngegneria s = new StudenteIngegneria();
    s.setNome("Mario");
    s.setCognome("Rossi");
    s.setMatricola("09204167");
    s.setTipo("Informatica");
    System.out.println("Nome: " + s.getNome());
    System.out.println("Cognome: " + s.getCognome());
    System.out.println("Matricola: " + s.getMatricola());
    System.out.println("Ingegneria " + s.getTipo());
  }

}
```

Il risultato prodotto è facilmente intuibile:

```
Nome: Mario
Cognome: Rossi
Matricola: 09204167
Ingegneria Informatica
```

La gerarchia implementata è così rappresentabile:

```
Object
  |
  +- Persona
       |
       +- Studente
            |
            +- StudenteIngegneria
```

Un oggetto di tipo *StudenteIngegneria* ha:

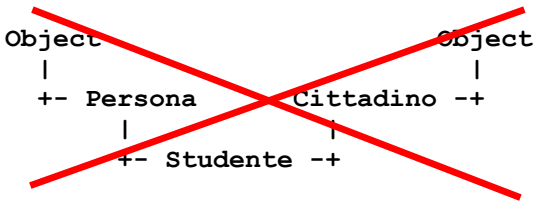
1. Il campo *tipo*, ed i metodi ad esso associati.
2. Il campo *matricola* ed i metodi ad esso associati, poiché li eredita da *Studente*.
3. I campi *nome* e *cognome*, con i metodi ad essi associati, poiché li eredita da *Studente* che li ha ereditati da *Persona*.
4. Tutti i campi ed i metodi di *Object*, che gli sono arrivati percorrendo l'intera gerarchia delle classi.

Ogni classe può avere avere infiniti figli. Quindi, è sempre possibile uno schema come il seguente:

```
  Object
    |
    +- Persona
         |   |
Operaio -+   +- Studente
```

In questo caso, non c'è rapporto diretto tra *Operaio* e *Studente*, se non nel fatto che ambo le classi derivano da *Persona*. Quindi, *Operaio* avrà tutte le caratteristiche di *Persona*, così come le avrà *Studente*. I nuovi campi campi definiti da *Operaio*, ad ogni modo, non saranno condivisi da *Studente*, e viceversa. Insomma, in questo punto la gerarchia si divide in più rami indipendenti. Le classi, procedendo avanti in un albero di questo tipo, saranno parenti sempre più alla lontana.

Una classe può avere un solo genitore. Il seguente schema, in Java, non è corretto:



Non è lecito che *Studente* possa contemporaneamente derivare sia da *Persona* sia da *Cittadino*. In Java, l'ereditarietà multipla non è ammessa. La nota si rivolge, in particolar modo, ai programmatori provenienti dall'ambito di C++, dove è possibile far ricorso all'ereditarietà multipla. I progettisti di Java, quando svilupparono il linguaggio, conclusero che l'ereditarietà multipla crei più problemi di quanti non ne risolva. In compenso, una sorta di

ereditarietà multipla è concessa attraverso le *interfacce*, che saranno esaminate nella prossima lezione.

10.2 - Impedire l'ereditarietà

Quando si realizza una classe, si può desiderare che questa non possa essere successivamente derivata. Per raggiungere lo scopo è possibile ricorrere alla parola chiave *final*, usata al seguente modo:

```
final class NomeClasse {
  // ...
}
```

Una classe dichiarata *final* non potrà avere sottoclassi. La verifica è immediata. Si riprenda la classe *Persona*, rendendola *final*:

```
final class Persona {

  String nome;
  String cognome;

  public void setNome(String nome) {
    this.nome = nome;
  }

  public void setCognome(String cognome) {
    this.cognome = cognome;
  }

  public String getNome() {
    return nome;
  }

  public String getCognome() {
    return cognome;
  }

}
```

La classe *Studente* estende *Persona*:

```
class Studente extends Persona {

  String matricola;

  public void setMatricola(String matricola) {
    this.matricola = matricola;
  }

  public String getMatricola() {
    return matricola;
  }

}
```

Non è più possibile compilare *Studente*:

```
Studente.java:1: cannot inherit from final Persona
public class Studente extends Persona {
                              ^
```

10.3 - Ereditarietà ed accesso ai membri

Ogni sottoclasse contiene sempre tutti i membri, cioè i metodi e le proprietà, della sua superclasse, tuttavia non è detto che possa sempre accedervi. Java ammette quattro differenti specificatori d'accesso per i membri di una classe:

1. ***public***. I membri pubblici non soffrono restrizioni. Chiunque può accedervi, da qualsiasi punto del software.
2. ***private***. I membri privati possono essere sfruttati solo ed esclusivamente all'interno della classe che li definisce. Una sottoclasse non può accedere ai membri privati della sua superclasse.
3. ***protected***. I membri protetti sono simili a quelli privati, con l'eccezione che le sottoclassi e le classi appartenenti al medesimo pacchetto possono accedervi liberamente.
4. **nessuno specificatore**. L'assenza di specificatore porta ad una condizione mista. Il membro si comporta come se fosse pubblico nei confronti di tutte le classi contenute nello stesso pacchetto, mentre appare privato all'esterno di esso.

I pacchetti saranno adeguatamente trattati nel corso della dodicesima lezione di questo corso, dunque ogni discorso legato al tema riceverà il giusto grado di approfondimento. Comunque sia, si prenda sempre a riferimento il seguente specchietto riassuntivo.

	public	***protected***	***niente***	***private***
Stessa classe	✔	✔	✔	✔
Sottoclasse stesso pacchetto	✔	✔	✔	✘
Non sottoclasse stesso pacchetto	✔	✔	✔	✘
Sottoclasse altro pacchetto	✔	✔	✘	✘
Non sottoclasse altro pacchetto	✔	✘	✘	✘

Tabella 10.1
Specificatori d'accesso. Da dove si ha libero accesso ai membri di una classe?

Si prenda nuovamente in esame la classe *Persona*, con una piccola ma significativa modifica:

```
class Persona {

  private String nome;
  private String cognome;

  public void setNome(String nome) {
    this.nome = nome;
```

```
  }

  public void setCognome(String cognome) {
    this.cognome = cognome;
  }

  public String getNome() {
    return nome;
  }

  public String getCognome() {
    return cognome;
  }

}
```

Le proprietà *nome* e *cognome*, all'interno di questa revisione, sono state rese ad accesso privato. Nessuno, all'infuori del codice di *Persona*, potrà accedervi in maniera diretta. La regola ha valore anche nel caso della classe derivata *Studente*. Ecco un test:

```
class Studente extends Persona {

  String matricola;

  public void setMatricola(String matricola) {
    this.matricola = matricola;
  }

  public String getMatricola() {
    return matricola;
  }

  public void stampaInformazioni() {
    System.out.println(nome);
    System.out.println(cognome);
    System.out.println(matricola);
  }

}
```

Per fini dimostrativi, la classe *Studente* è stata dotata di un nuovo metodo, *stampaInformazioni()*, il cui compito è riassumere tutte le proprietà dell'oggetto di invocazione. In virtù della modifica effettuata su *Persona*, è impossibile compilare la classe *Studente*. Il doppio errore che si riceve è:

```
Studente.java:14: nome has private access in Persona
            System.out.println(nome);
                   ^
Studente.java:15: cognome has private access in Persona
            System.out.println(cognome);
                   ^
```

Come dare torto al compilatore? Le stringhe *nome* e *cognome*, in effetti, hanno accesso privato, riservato al solo codice di *Persona*. Dunque, neanche una sottoclasse di *Persona*,

come lo è *Studente*, può accedervi. In compenso, è possibile sfruttare i metodi *getNome()* e *getCognome()*, che sono pubblici. Basterebbe ridefinire *stampaInformazioni()* alla seguente maniera:

```
public void stampaInformazioni() {
  System.out.println(getNome());
  System.out.println(getCognome());
  System.out.println(matricola);
}
```

Gli specificatori d'accesso sono un potente strumento. Permettono di limitare l'uso che sarà fatto dei membri di una classe, in tre differenti situazioni:

1. Quando si impiega la classe per i suoi scopi finali.
2. Quando si deriva la classe.
3. Quando si impiega o si deriva la classe per completare con altri elementi il pacchetto che la contiene.

10.4 - Ereditarietà e costruttori

Finora, si è parlato di ereditarietà, ma non sono ancora stati presi in considerazione i rapporti che corrono tra i costruttori di una superclasse e quelli delle sue sottoclassi.

Regola numero 1: i costruttori non vengono ereditati. Se una classe *A* dispone di un costruttore con tre argomenti di tipo *String*, giusto per citare un esempio, una sua sottoclasse *B* non disporrà automaticamente dello stesso costruttore. Vale sempre la medesima norma, anche nei casi di ereditarietà: se nessun costruttore è esplicitamente dichiarato, la classe verrà automaticamente dotata di un costruttore di default, privo di argomenti, che non compie alcuna operazione. Al contrario, se si definisce anche un solo costruttore, quello di default privo di argomenti non sarà più fornito automaticamente dal linguaggio.

Regola numero 2: quando si istanzia una sottoclasse, almeno uno dei costruttori della sua superclasse deve essere richiamato implicitamente o esplicitamente. Andiamo per casi. Quando si istanzia una sottoclasse, mediante uno qualsiasi dei suoi costruttori, viene automaticamente richiamato il costruttore senza argomenti della sua superclasse. La verifica è semplice:

```
class A {

  public A() {
    System.out.println("Costruttore di A");
  }

}

class B extends A {

  public B() {
    System.out.println("Costruttore di B");
  }
```

```
}
```

Si esegua il seguente test, che usa le due classi appena definite:

```
class Test {

  public static void main(String[] args) {
    B b = new B();
  }

}
```

L'output prodotto è il seguente:

```
Costruttore di A
Costruttore di B
```

Come è possibile osservare, un oggetto di tipo *B* è stato costruito eseguendo prima il costruttore della superclasse *A* e poi quello della sottoclasse *B*, in questo preciso ordine. La norma vale finché la superclasse dispone di un costruttore privo di argomenti. In caso contrario, è necessario specificare esplicitamente quale dei costruttori della superclasse vada richiamato, all'interno di ogni costruttore della sottoclasse realizzata. A tal fine, può essere usata la parola chiave ***super***:

```
class A {

  private int x;

  public A(int x) {
    this.x = x;
  }

  public int getX() {
    return x;
  }

}

class B extends A {

  private int y;

  public B(int x, int y) {
    super(x);
    this.y = y;
  }

  public int getY() {
    return y;
  }

}

class Test {
```

```
  public static void main(String[] args) {
    B b = new B(5, 7);
    System.out.println("b.x = " + b.getX());
    System.out.println("b.y = " + b.getY());
  }

}
```

In *A* è stato definito un solo costruttore, che accetta un parametro di tipo *int*. In questo modo, nessun costruttore privo di argomenti è più disponibile. In *B* è stato introdotto un costruttore con due argomenti di tipo *int*. Il primo di essi, *x*, viene fornito al costruttore della superclasse *A*, attraverso una chiamata a *super*:

```
super(x);
```

Eseguendo la classe *Test*, si ottiene il seguente output, che rispetta le aspettative:

```
b.x = 5
b.y = 7
```

Alcune precisazioni sulla parola chiave super, nel contesto dei costruttori:

1. *super*, se impiegata, deve essere sempre la prima istruzione presente in un costruttore.
2. Quando la superclasse dispone di più costruttori, compreso o meno quello privo di argomenti, l'istruzione *super* può essere usata per stabilire a quale di essi appellarsi.

Un ulteriore esempio, ottenuto revisionando le classi *Persona* e *Studente* esaminate in precedenza:

```
class Persona {

  private String nome;
  private String cognome;

  public Persona(String nome, String cognome) {
    this.nome = nome;
    this.cognome = cognome;
  }

  public String getNome() {
    return nome;
  }

  public String getCognome() {
    return cognome;
  }

}
```

In questa versione di *Persona*, sono stati rimossi i metodi *setNome()* e *setCognome()*. Ora è

possibile impostare i campi *nome* e *cognome* direttamente alla creazione di un oggetto *Persona*, servendosi del costruttore della classe. Fin qui, nessun problema. Ma cosa accade nella sottoclasse *Studente*? Si torni a considerarla nella sua formulazione originaria:

```
class Studente extends Persona {

  String matricola;

  public void setMatricola(String matricola) {
    this.matricola = matricola;
  }

  public String getMatricola() {
    return matricola;
  }

}
```

La compilazione non riesce. L'errore è il seguente:

```
Studente.java:1: cannot resolve symbol
symbol  : constructor Persona  ()
location: class Persona
public class Studente extends Persona {
       ^
```

La classe *Studente* non ha costruttore. Pertanto, il compilatore gliene fornisce uno di default, privo di argomenti, che non compie alcuna operazione. Al suo interno, quindi, non ci sarà nessuna chiamata a *super*. Ne consegue che, alla creazione di un oggetto *Studente*, sarà richiesto un costruttore di *Persona* privo di argomenti. Giacché non esiste un costruttore di questo tipo, l'operazione è impossibile. Per questo, il compilatore interrompe il proprio operato, impedendo anzitempo la catastrofe. Quindi, o si dota *Persona* di un costruttore privo di argomenti, oppure si definisce un costruttore in *Studente* che faccia uso di *super*. Si opti per la seconda soluzione, che tecnicamente è quella più corretta:

```
class Studente extends Persona {

  String matricola;

  public Studente(String nome, String cognome, String matricola) {
    super(nome, cognome);
    this.matricola = matricola;
  }

  public void setMatricola(String matricola) {
    this.matricola = matricola;
  }

  public String getMatricola() {
    return matricola;
  }

}
```

Missione compiuta! Da questo momento in poi, si potranno creare istanze di *Studente* ricalcando il seguente modello:

```
Studente s = new Studente("Mario", "Rossi", "09204167");
```

10.5 - Ridefinizione dei metodi

Quando il metodo di una sottoclasse ha la stessa firma di un metodo della sua superclasse, si dice che il metodo della sottoclasse *ridefinisce* (*override*, in inglese) quello della superclasse. Ogni chiamata a tale metodo farà riferimento alla versione ridefinita. Il meccanismo è molto semplice, e si può dimostrare facilmente:

```
class A {

  public void prova() {
    System.out.println("prova() di A");
  }

}

class B extends A {

  public void prova() {
    System.out.println("prova() di B");
  }

}

class Test {

  public static void main(String[] args) {
    A a = new A();
    B b = new B();
    a.prova();
    b.prova();
  }

}
```

Eseguendo il test, si otterrà l'output:

```
prova() di A
prova() di B
```

Nella classe *A* è definito il metodo *prova()*. Ovviamente, nel momento in cui si crea un'istanza di *A*, la chiamata al metodo darà sempre l'output:

```
prova() di A
```

Non potrebbe essere diversamente. La classe *B* estende *A* e ridefinisce *prova()*. Chiamando il metodo su un'istanza di *B*, si ottiene:

```
prova() di B
```

Quindi, la ridefinizione contenuta in *B* ha nascosto l'originaria formulazione presente in *A*.

Un secondo utilizzo di *super* consente, dall'interno di una sottoclasse, l'accesso ad un metodo nascosto della superclasse. Esaminiamo un esempio:

```
class A {

  public void prova() {
    System.out.println("prova() di A");
  }

}

class B extends A {

  public void prova() {
    super.prova();
    System.out.println("prova() di B");
  }

}

class Test {

  public static void main(String[] args) {
    B b = new B();
    b.prova();
  }

}
```

In questo caso, il metodo *prova()* di *B* si appella al suo omonimo nella superclasse *A*, quindi l'output sarà:

```
prova() di A
prova() di B
```

Nei casi di ridefinizione, l'utilizzo di *super* è molto più libero che non nel contesto dei costruttori. Un metodo nascosto può essere richiamato da qualsiasi punto, all'interno della sottoclasse che lo ridefinisce.

Quando si progetta una classe, si può fare in modo che alcuni dei suoi metodi non possano essere ridefiniti da eventuali sottoclassi. La parola chiave utile per raggiungere tale scopo è la poliedrica *final*:

```
class A {

  public final void prova() {
    System.out.println("prova() di A");
  }
```

```
}

class B extends A {

  public void prova() {
    System.out.println("prova() di B");
  }

}
```

Il metodo *prova()*, all'interno di *A*, è stato dichiarato *final*. *B*, ora, non ha più la facoltà di ridefinirlo. Tendando la compilazione di *B*, si otterrà il seguente errore:

```
prova() in B cannot override prova() in A; overridden method is final
```

10.6 - Utilizzo delle classi derivate

Un aspetto che rende utile la pratica dell'ereditarietà consiste nel fatto che una variabile che fa riferimento ad una superclasse può fare riferimento anche ad una sua qualsiasi sottoclasse. Detto a parole sembra complicato, anche se in realtà non lo è. Un codice esemplificativo sarà d'aiuto:

```
class A {

  public void prova() {
    System.out.println("prova() di A");
  }

}

class B extends A {

  public void prova() {
    System.out.println("prova() di B");
  }

}

class C extends B {

  public void prova() {
    System.out.println("prova() di C");
  }

}

class Test {

  public static void main(String[] args) {
    A a1 = new A();
    A a2 = new B();
    A a3 = new C();
    a1.prova();
    a2.prova();
    a3.prova();
  }
```

```
}
```

Ci sono tre classi, chiamate *A*, *B* e *C*, disposte secondo la seguente gerarchia:

```
Object
  |
  +- A
     |
     +- B
        |
        +- C
```

Nella classe di verifica *Test*, si eseguono le seguenti dichiarazioni:

```
A a1 = new A();
A a2 = new B();
A a3 = new C();
```

Sono state dichiarate tre variabili, tutte di tipo *A*. In *a1*, come consuetudine, è stato memorizzato il riferimento ad un oggetto di tipo *A*. Fin qui, nulla di nuovo. In *a2* e *a3*, invece, sono state memorizzate delle istanze di *B* e di *C*. Ciò è possibile grazie alla norma illustrata in apertura di paragrafo. Successivamente, è stato invocato il metodo *prova()* su ogni istanza creata, ottenendo l'output:

```
prova() di A
prova() di B
prova() di C
```

Nonostante il riferimento tenuto sia di tipo *A*, ad ogni istanza viene comunque associato il metodo competente. Il motore di runtime esegue uno smistamento automatico dei metodi ridefiniti. La caratteristica è molto utile quando si fa ricorso ad array o collezioni, come nel seguente caso:

```
class Test2 {

  public static void main(String[] args) {
    A[] a = new A[3];
    a[0] = new A();
    a[1] = new B();
    a[2] = new C();
    for (int i = 0; i < a.length; i++)
      a[i].prova();
  }

}
```

Al contrario, non è possibile richiamare i metodi specifici di una sottoclasse passando per un riferimento alla sua superclasse, come nel seguente caso:

```
class A {
```

```
  public void prova() {
    System.out.println("prova() di A");
  }

}

class B extends A {

  public void prova() {
    System.out.println("prova() di B");
  }

  public void esegui() {
    System.out.println("esegui()");
  }

}

class Test {

  public static void main(String[] args) {
    A a = new B();
    a.esegui();
  }

}
```

B definisce un metodo chiamato *esegui()*, che non è proprio di *A*. Quindi, non si tratta di una ridefinizione. La compilazione della classe *Test* fallisce, poiché non è lecito richiamare *esegui()* su una variabile di tipo *A*, anche nel caso in cui il riferimento in essa contenuto conduca ad un'istanza di *B*. Il messaggio di errore che si riceve è:

```
Test.java:5: cannot resolve symbol
symbol  : method esegui  ()
location: class A
    a.esegui();
      ^
```

In condizioni come quella appena presentata, il casting (o conversione esplicita) è un'ottima maniera per ottenere un riferimento alla sottoclasse, sul quale poi richiamare il metodo desiderato. La spiegazione viene esplicitata nel seguente esempio di impiego delle classi *A* e *B*:

```
class Test2 {

  public static void main(String[] args) {
    A a = new B();
    B b = (B)a;
    b.esegui();
  }

}
```

In casi di questo tipo, inoltre, torna particolarmente utile l'operatore *instanceof*. Si utilizza al

seguente modo:

```
riferimento instanceof NomeClasse
```

L'espressione restituisce *true* se il riferimento usato come primo operando conduce ad un'istanza della classe posta al secondo operando. In caso contrario, ovviamente, si otterrà un bel *false*. Usando sempre le classi *A*, *B* e *C*, è possibile verificare la funzionalità dell'operatore:

```
public class A {}

public class B extends A {}

public class C extends B {}

public class Test {

  public static void main(String[] args) {
    A a1 = new A();
    A a2 = new B();
    A a3 = new C();
    System.out.println(
      "a1 instanceof A: " + (a1 instanceof A)
    );
    System.out.println(
      "a1 instanceof B: " + (a1 instanceof B)
    );
    System.out.println(
      "a1 instanceof C: " + (a1 instanceof C)
    );
    System.out.println();
    System.out.println(
      "a2 instanceof A: " + (a2 instanceof A)
    );
    System.out.println(
      "a2 instanceof B: " + (a2 instanceof B)
    );
    System.out.println(
      "a2 instanceof C: " + (a2 instanceof C)
    );
    System.out.println();
    System.out.println(
      "a3 instanceof A: " + (a3 instanceof A)
    );
    System.out.println(
      "a3 instanceof B: " + (a3 instanceof B)
    );
    System.out.println(
      "a3 instanceof C: " + (a3 instanceof C)
    );
  }

}
```

L'output che si riceve è il seguente:

```
a1 instanceof A: true
a1 instanceof B: false
a1 instanceof C: false

a2 instanceof A: true
a2 instanceof B: true
a2 instanceof C: false

a3 instanceof A: true
a3 instanceof B: true
a3 instanceof C: true
```

La risposta ottenuta è corretta:

1. *a1* è istanza di *A*, ma non di *B* e *C*.
2. *a2* è istanza di *B*, e poiché *B* eredita da *A*, è anche istanza di *A*. Non è, però, istanza di *C*.
3. *a3* è istanza di *C*, e per ereditarietà è istanza anche di *B* e di *A*.

10.7 - Classi astratte

Le classi astratte sono classi create appositamente per essere derivate. Una classe astratta può essere derivata, ma non è possibile avere delle sue istanze da impiegare direttamente in un software. Alla base dell'astrazione di una classe è la parola chiave ***abstract***. Una classe, pertanto, è astratta quando è dichiarata secondo il modello:

```
abstract class {
  // ...
}
```

Le classi astratte contengono sempre uno o più metodi astratti. Un metodo è astratto quando:

1. La sua dichiarazione è preceduta dalla parola chiave *abstract*.
2. Non ha corpo.

Le classi astratte sono impiegate per definire un modello di comportamento, che ogni sottoclasse dovrà rispettare, implementando tutti i metodi astratti in essa contenuti. Si prenda in esame la classe astratta *Animale*:

```
abstract class Animale {

  public void mangia() {
    System.out.println("Gnam!");
  }

  public abstract void faiVerso();

}
```

La classe contiene due metodi: *mangia()* e *faiVerso()*. L'implementazione del primo è molto semplice da realizzare. Ma cosa si deve mettere dentro *faiVerso()*? *Animale* è una classe che

esprime un concetto molto generico e assai poco concreto. Che verso fa un animale? La risposta dipende dal tipo di animale. Un gatto fa "miao", un cane "bau"... ma un animale in senso generico? Non è possibile stabilirlo! Per questo motivo, la classe *Animale* è stata dichiarata astratta, ed il suo metodo *faiVerso()* non è stato definito. Di conseguenza, non è possibile istanziare oggetti della classe *Animale*.

```
Animale a = new Animale();
```

Un'istruzione come questa porta ad un errore di compilazione:

```
Animale is abstract; cannot be instantiated
```

Animale esiste solo per essere derivata. Ogni suo sottoclasse dovrà obbligatoriamente dare definizione al metodo *faiVerso()*, tranne nel caso non sia essa stessa una classe astratta, pena un errore di compilazione. Ecco alcune simpatiche implementazioni:

```
class Gatto extends Animale {

  public void faiVerso() {
    System.out.println("Miaaaaaaaaooo!");
  }

}

class Cane extends Animale {

  public void faiVerso() {
    System.out.println("Bau!");
  }

}

class Topo extends Animale {

  public void faiVerso() {
    System.out.println("Squit!");
  }

}
```

Si verifichi la funzionalità delle tre classi appena realizzate:

```
class Test {

  public static void main(String[] args) {
    Animale[] zoo = new Animale[3];
    zoo[0] = new Gatto();
    zoo[1] = new Cane();
    zoo[2] = new Topo();
    for (int i = 0; i < zoo.length; i++)
      zoo[i].faiVerso();
  }
```

```
}
```

L'output è il seguente:

```
Miaaaaaaaaooo!
Bau!
Squit!
```

Come sarà spiegato nella prossima lezione, le interfacce sono una completa estremizzazione del concetto di classe astratta.

10.8 - Ereditarietà: perché?

Spesso l'ereditarietà è considerata uno dei concetti più ostici della programmazione orientata agli oggetti. Più che altro, questo avviene poiché i semplici esempi iniziali che si possono dare in una lezione come questa, non sono sufficientemente significativi da inquadrare la potenza astrattiva del meccanismo. Quindi, un avviso: non si sottovalutino i concetti appena esaminati, giacché torneranno presto preziosi! Impiegandoli con parsimonia, si potranno realizzare progetti sicuri, ben organizzati, scalabili e comprensibili, come la più pura programmazione orientata agli oggetti desidera. Inoltre, la piena comprensione dell'ereditarietà permette un uso consapevole delle librerie base di Java, che spesso e volentieri fanno ricorso a questo meccanismo.