Linguaggi di Programmazione I – Lezione 13

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

22 maggio 2008

Classi astratte e interfacce

Classi astratte

Interfacce

Esercizi

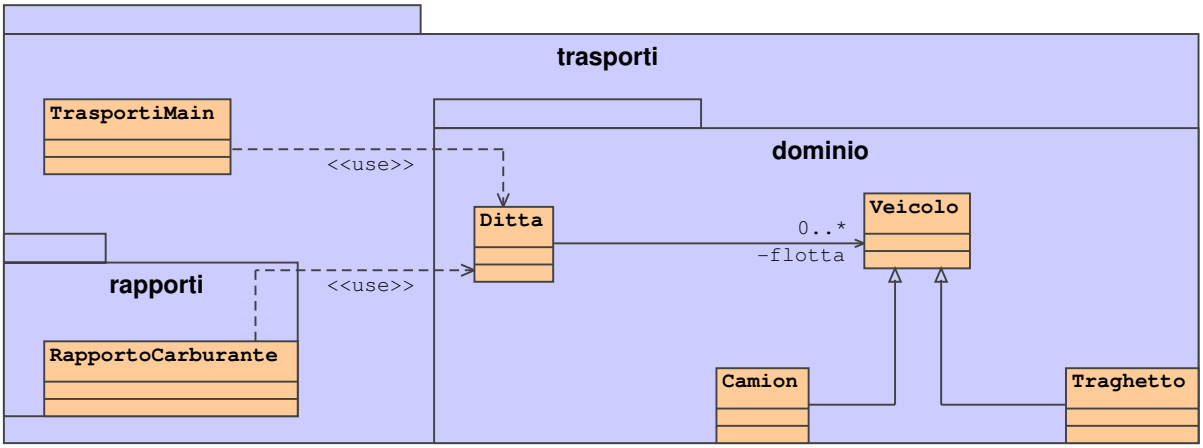
Casting di riferimenti

Questionario

Classi astratte

Intro (1)

Supponiamo che si voglia un rapporto settimanale sul consumo dei veicoli di una ditta di trasporti.
Il diagramma UML potrebbe essere questo:



Il codice che realizza le due classi esterne al `dominio` sia:

Intro (2)

```
public class TrasportiMain {
  public static void main(String[] args) {
    Ditta d = Ditta.getDitta(); // Singoletto

    // popola la flotta di veicoli
    d.addVeicolo(new Camion(10000.0));
    d.addVeicolo(new Camion(15000.0));
    d.addVeicolo(new Traghetto(500000.0));
    d.addVeicolo(new Camion(9500.0));
    d.addVeicolo(new Traghetto(750000.0));

    RapportoCarburante rapporto = new RapportoCarburante();
    rapporto.generaTesto(System.out);
  }
}
```

Intro (3)

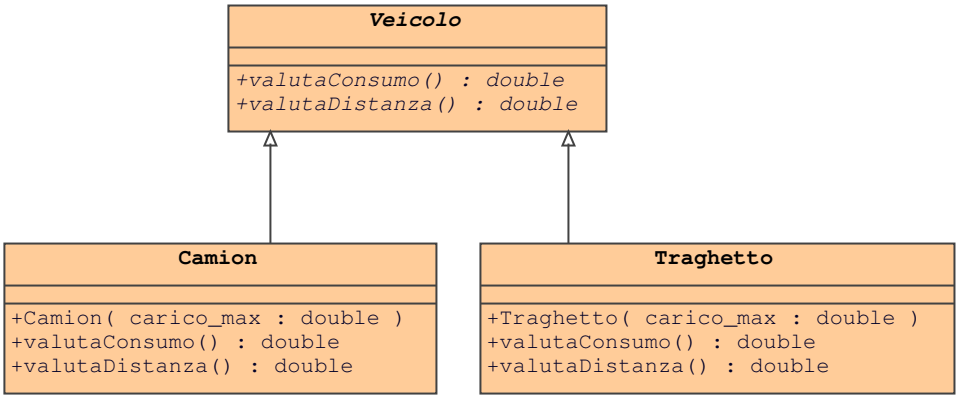
```
public class RapportoCarburante {
  public void generaTesto(PrintStream output) {
    Ditta d = Ditta.getDitta(); // Singoletto
    Veicolo v;
    double carburante;
    double totale = 0.0;

    for (int i=0; i<d.getDimFlotta(); i++) {
      v = d.getVeicolo(i);
      carburante = v.valutaConsumoPerKm() * v.valutaDistanza();
      output.println("Consumo del veicolo " + v.getNome() +
                     ": " + carburante);
      totale += carburante;
    }
    output.println("Consumo totale: ", totale);
  }
}
```


Problema A

- La valutazione dei consumi tra i due tipi di veicoli potrebbe essere totalmente differente.
- Non ha senso che la classe `Veicolo` fornisca i due metodi `valutaDistanza()` e `valutaConsumoPerKm()`, ma le sue sottoclassi si.

Soluzione A.1

- Java permette ad una superclasse di dichiarare un metodo, del quale non fornirà una realizzazione.
- La realizzazione (*implementazione*) sarà fornita dalle sottoclassi.
- Tali metodi sono *astratti* e una classe con uno o più metodi astratti è una *classe astratta*.



- In UML ricordiamo che metodi o classi astratte si rappresentano utilizzando il carattere *corsivo*.
- Java proibisce la costruzione di oggetti di una classe astratta.
- Tuttavia, classi astratte possono avere attributi, metodi concreti e costruttori. È buona norma rendere questi costruttori `protected` piuttosto che `public` (perché?).

Soluzione A.2

```
public abstract class Veicolo {
  public abstract double valutaConsumoPerKm();
  public abstract double valutaDistanza();
}
```

```
public class Camion extends Veicolo {
  public Camion (double carico_max) {...}

  public double valutaConsumoPerKm() {
    // Restituisce il consumo/Km
    // per un certo modello di camion
  }
  public double valutaDistanza() {
    // Restituisce la distanza che deve
    // percorrere per i viaggi
  }
}
```


Soluzione A.3

```
public class Traghetto extends Veicolo {
  public Traghetto (double carico_max) {...}

  public double valutaConsumoPerKm() {
    // Restituisce il consumo/Km
    // per un certo traghetto
  }
  public double valutaDistanza() {
    // Restituisce la distanza che deve
    // percorrere per i viaggi in mare
  }
}
```

Problema B

Riprendiamo la classe `RapportoCarburante`:

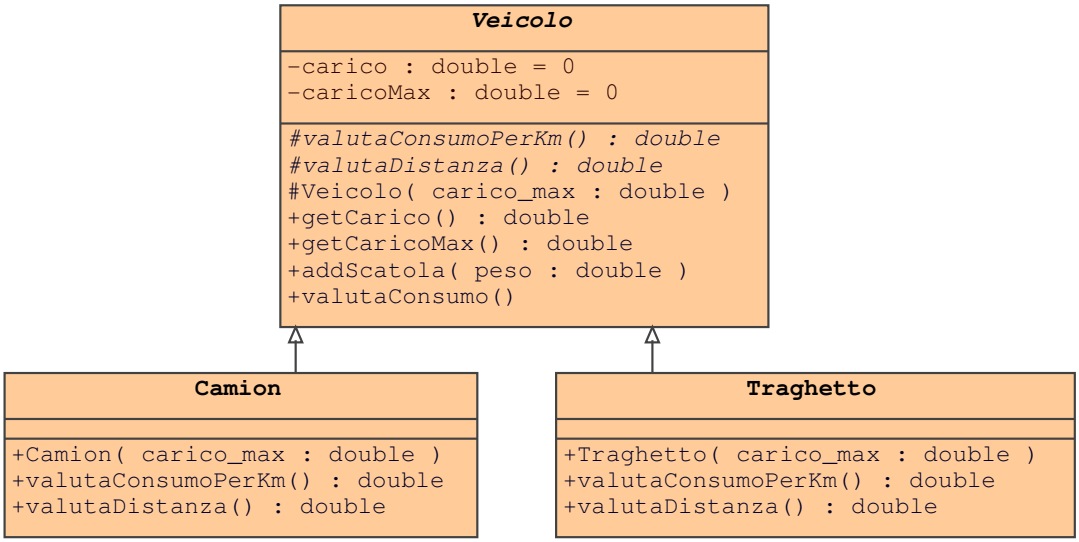
```
public class RapportoCarburante {
  public void generaTesto(PrintStream output) {
    ...
      carburante = v.valutaConsumoPerKm() * v.valutaDistanza();
    ...
  }
}
```

Il calcolo del consumo di un veicolo non dovrebbe essere eseguito in questa classe, ma dovrebbe essere una responsabilità del veicolo stesso, cioè appartenere alla classe `Veicolo`. Ma nella classe veicolo non ci sono le realizzazioni dei metodi `valutaConsumoPerKm()` e `valutaDistanza()`.
Nessun problema, perché l'invocazione virtuale dei metodi serve proprio a questo. Deleghiamo, pertanto, ad un veicolo il calcolo del consumo:

```
public class RapportoCarburante {
  public void generaTesto(PrintStream output) {
    ...
      carburante = v.valutaConsumo();
    ...
  }
}
```


Soluzione B

- La tecnica di soluzione consiste nel porre in una classe astratta un metodo concreto (nel nostro caso `valutaConsumo()`) che utilizzi metodi astratti nella stessa classe.
- Poiché tale tecnica è assai ricorrente, essa viene comunemente indicata come *Metodo Sagoma* (*Template Method Design Pattern*).



Generalità

- Una "interfaccia" è un contratto tra il codice cliente e la classe che "implementa" quell'interfaccia.
- In Java è una formalizzazione di un tale contratto in cui tutti i metodi non contengono realizzazioni.
- Molte classi, anche non correlate, possono implementare la stessa interfaccia.
- Una classe può implementare molte interfacce, anche non correlate.
- Una interfaccia può estendere altre interfacce.

Sintassi

- La sintassi è:

```
<class_declaration> ::=
  <modifier> class <name> [extends <superclass>]
      [implements <interface> [, <interface>]* ] {
    <class_body>
  }

<interface_declaration> ::=
  <modifier> interface <name>
      [extends <interface> [, <interface>]* ] {
    <interface_body>
  }
```

- Nota – una interfaccia può anche dichiarare costanti:

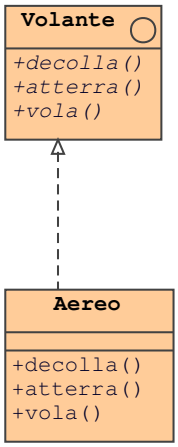
```
public static final int COSTANTE = 7;
```

Attenzione: le variabili dichiarate nelle interfacce sono implicitamente `public static final`. Pertanto la dichiarazione precedente poteva anche essere scritta:

```
int COSTANTE = 7;
```

Esempio (1)

Immaginiamo un gruppo di oggetti che condividono la stessa abilità: essi volano.
Si può costruire una interfaccia pubblica, chiamata `Volante`, che descriva tre operazioni: `decolla`, `atterra` e `vola`.

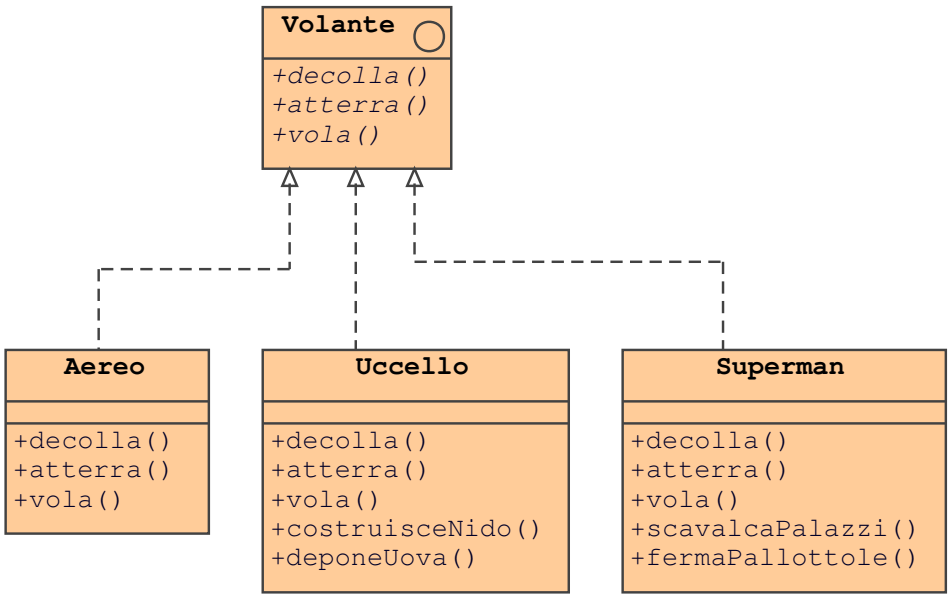



```
public interface Volante {
  public void decolla();
  public void atterra();
  public void vola();
}
```

```
public class Aereo
    implements Volante {
  public void decolla() {
    // accelera fino a sollevarsi
    // alza il carrello
  }
  public void atterra() {
    // abbassa il carrello
    // decelera e abbassa i flap
    // fino a toccare terra, poi
    // frena
  }
  public void vola() {
    // tieni il motore acceso
  }
}
```

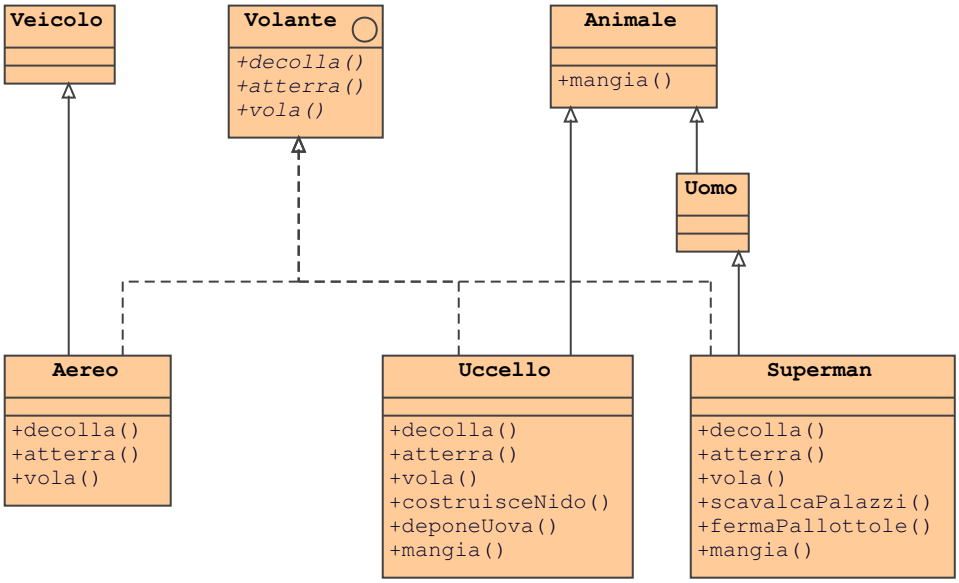

Esempio (2)

- Ma quali altri oggetti volano?
- Per esempio:



Esempio (3)

Qual è la superclasse di `Uccello`? Che cos'è un uccello?



Questo suona come ereditarietà multipla. Non proprio. Il pericolo dell'ereditarietà multipla è che una classe possa ereditare due distinte implementazioni dello stesso metodo. Ciò non è possibile con le interfacce.

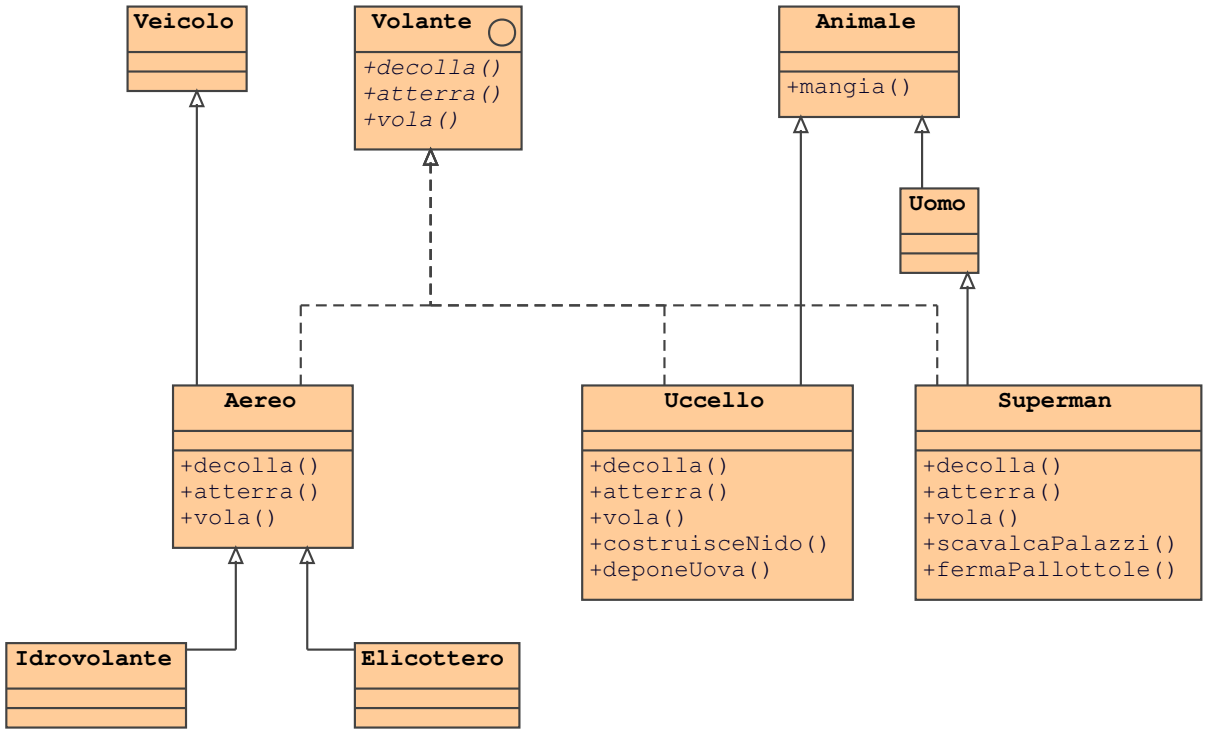
Esempio (4)

Prendiamo per esempio la classe `Uccello`:

```
public class Uccello extends Animale implements Volante {
  public void decolla() {
    // implementazione di un metodo
  }
  public void atterra() {
    // implementazione di un metodo
  }
  public void vola() {
    // implementazione di un metodo
  }
  public void costruisceNido() {
    // competenza propria
  }
  public void deponeUova() {
    // competenza propria
  }
  public void mangia() {
    // sovrapposizione di un metodo
  }
}
```

Esempio (5)

Supponiamo di voler costruire un sistema software di controllo dei voli. Esso dovrà garantire i permessi di decollo e di atterraggio a qualunque tipo di oggetto volante.



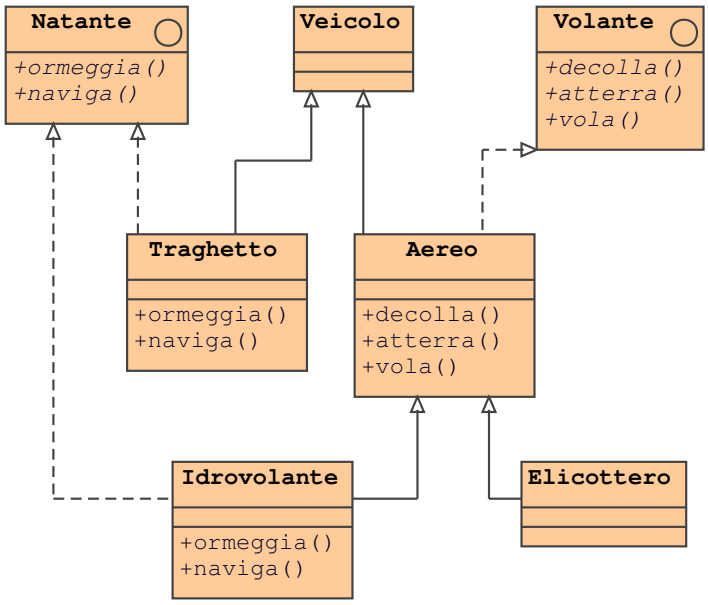
Esempio (6)

Una classe che userà le classi precedenti potrebbe, per esempio, essere:

```
public class Aeroporto {
  public static void main(String[] args) {
    Aeroporto metropolis = new Aeroporto();
    Elicottero eli = new Elicottero();
    Idrovolante idro = new Idrovolante();
    Volante S = Superman.getSuperman();

    metropolis.daiPermessoAtterraggio(eli);
    metropolis.daiPermessoAtterraggio(idro);
    metropolis.daiPermessoAtterraggio(S);
  }

  private void daiPermessoAtterraggio(Volante v) {
    v.atterra();
  }
}
```

Esempio (7)

Una classe può implementare più di una interfaccia. Un idrovolante, non solo vola, ma anche naviga. La classe `Idrovolante` estende la classe `Aereo`, ed eredita l'implementazione dell'interfaccia `Volante`, ma implementa anche l'interfaccia `Natante`.



Esempio (8)

In quest'ultimo caso, una classe che userà le classi precedenti potrebbe essere:

```
public class Porto {
  public static void main(String[] args) {
    Porto portoNapoli = new Porto();
    Traghetto trag = new Traghetto();
    Idrovolante idro = new Idrovolante();

    portoNapoli.daiPermessoOrmeggio(trag);
    portoNapoli.daiPermessoOrmeggio(idro);
  }

  private void daiPermessoOrmeggio(Natante n) {
    n.ormeggia();
  }
}
```

Vantaggi delle interfacce

Le interfacce[a] sono spesso considerate una alternativa all'ereditarietà multipla, anche se esse forniscono diversa funzionalità.
Esse sono utili:

- per dichiarare metodi che una o più classe ci si aspetta dovrà implementare;
- per rivelare solo una ... interfaccia, senza rivelare il corpo vero di una classe (per esempio quando si distribuisce una classe ad altri sviluppatori di codice);
- per catturare similarità tra classi non correlate, senza forzare una relazione tra classi;
- per simulare l'ereditarietà multipla, dichiarando una classe che implementi varie interfacce.

[a] Il concetto di interfaccia è preso in prestito da Objective-C, in cui essa è chiamata *protocollo*.

Esercizi 25 / 44

Esercizi

1. Creare una gerarchia di animali con radice nella classe astratta `Animale`. Alcune delle classi dovranno implementare l'interfaccia `Domestico`.

Casting di riferimenti 26 / 44

Introduzione

- I riferimenti, come i primitivi, partecipano alla conversione automatica per assegnamento (e per passaggio di parametri) e al casting.
- Non c'è promozione aritmetica di riferimenti, poiché i riferimenti non possono essere operandi aritmetici.
- Nella conversione e nel casting di riferimenti vi sono maggiori combinazioni tra vecchi e nuovi tipi – e maggiori combinazioni significano un numero maggiore di regole.
- La conversione (automatica) di riferimenti avviene nella fase di compilazione, poiché il compilatore ha tutte le informazioni per determinare se la conversione è legale.
- Essa può essere forzata con un casting, come per i primitivi. In questo caso, può accadere che, sebbene la conversione sia autorizzata come legale dal compilatore, il tipo effettivo dell'oggetto riferito non sia compatibile in esecuzione (in una assegnazione entrano in gioco tre tipi: quello dei due riferimenti e quello proprio dell'oggetto; strano, ma vero).

Conversioni automatiche (1)

- Nella conversiona automatica di riferimenti (per ampliamento, in una assegnazione o in un passaggio di parametri), vengono nascoste alcune caratteristiche dell'oggetto riferito.
- Qui il tipo di nascita dell'oggetto non interessa; i riferimenti possono essere:
  - ad una classe;
  - ad una interfaccia;
  - ad un array.
- La conversione può avvenire in:

```
Oldtype x = new Oldtype();
Newtype y = x;
```

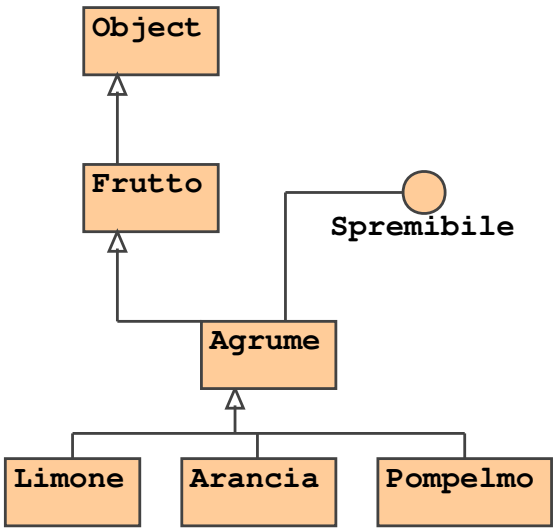
	`Oldtype` è una classe	`Oldtype` è una interfaccia	`Oldtype` è un array
`Newtype` è una classe	`Oldtype` deve essere sottoclasse di `Newtype`	`Newtype` deve essere `Object`	`Newtype` deve essere `Object`
`Newtype` è una interfaccia	`Oldtype` deve implementare l'interfaccia `Newtype`	`Oldtype` deve essere una sottointerfaccia di `Newtype`	`Newtype` deve essere `Cloneable` o `Serializable`
`Newtype` è un array	Errore di compilazione	Errore di compilazione	I tipi delle componenti dei due array devono essere riferimenti auto-convertibili

Conversioni automatiche (2)

Regola breve:

- Un tipo interfaccia può essere convertito solo ad un tipo interfaccia o ad `Object`. Se il nuovo tipo è una interfaccia essa deve essere una superinterfaccia di quella del vecchio tipo.
- Un tipo classe può essere convertito ad un tipo classe o ad un tipo interfaccia. Se convertito ad un tipo classe, il nuovo tipo deve essere una superclasse del vecchio tipo. Se convertito ad un tipo interfaccia, la vecchia classe deve implementare l'interfaccia.
- Un array può essere convertito alla classe `Object`, all'interfaccia `Cloneable` o all'interfaccia `Serializable`, oppure ad un array. Solo un array di riferimenti (non di primitivi) può essere convertito ad un array, e il vecchio tipo degli elementi deve essere convertibile al nuovo tipo degli elementi.

Esempi (1)




```
Arancia arancia = new Arancia();
Agrume agrume = arancia; // OK
```

```
Agrume agrume = new Agrume();
Arancia arancia = agrume; // NO
```

```
Pompelmo p = new Pompelmo();
Spremibile s = p; // OK
Pompelmo p2 = s; // Errore comp.
```

Una interfaccia può solo essere auto-convertita ad `Object`.

```
Frutto frutta[];
Limone limoni[];
Agrume agrumi[] = new Agrume[10];
for (int i=0; i<10; i++) {
  agrumi[i] = new Agrume();
}
frutta = agrumi; // OK
limoni = agrumi; // Errore comp.
```

In questo caso contano i tipi dei riferimenti negli elementi degli array.

Esempi (2)

A proposito di overloading e overriding, in attinenza alle conversioni automatiche di riferimenti, ricordiamo che la scelta del metodo da associare ad una invocazione avviene

- in compilazione per i metodi sovraccaricati (basandosi sul tipo dei riferimenti nei parametri),
- in esecuzione per i metodi sovrapposti (basandosi sulla effettiva presenza e visibilità del metodo nell'oggetto).

Quindi:

Esempi (3)

```
class A { }
class B extends A {
  public void g(A a) {
    System.out.println("A");
  }
}
class UseAB extends B {
  public void f(A a) {
    System.out.println("A");
  }
  public void f(B a) {
    System.out.println("B");
  }
  public void g(A b) {
    System.out.println("B");
  }

  public static void main (
                String[] args) {
    UseAB u = new UseAB();
    u.metodo();
  }
```

```
  public void metodo() {
    A a = new A();
    B b = new B();
    A x = b;        // auto-conv.

    f(a);           // stampa A
    f(b);           // stampa B
    f(x);           // stampa A!!!
    f(new B());     // stampa B

    g(a);           // stampa B
    g(b);           // stampa B
    g(x);           // stampa B
    g(new B());     // stampa B
    super.g(a);     // stampa A
    super.g(b);     // stampa A
    super.g(x);     // stampa A
  }

}
```


Casting (1)

Una volta appurato quali sono le conversioni che il compilatore accetta di fare da sè (quelle per assegnazione, o passaggio di parametri, di ampliamento), possiamo passare a studiare le conversioni che il programmatore chiede esplicitamente (casting).

- Un cast di ampliamento, la cui conversione sarebbe avvenuta anche senza di esso, viene autorizzato e non causa problemi nè in compilazione, nè in esecuzione.
- Detto rozzamente, un casting di riferimenti verso l'alto, nella gerarchia di ereditarietà, è sempre, sia implicitamente sia esplicitamente, legale.
- Purtroppo, per i riferimenti e a differenza dei primitivi, nel casting verso il basso, il compilatore non può aiutare completamente e può accadere che, sebbene la conversione sia autorizzata come legale, essa fallisca in esecuzione.
- Infatti, nel casting esplicito, entrano in gioco tre tipi: quello dei due riferimenti (a sinistra e a destra dell'assegnazione) e quello vero (l'identità, l'imprinting di nascita) dell'oggetto.
- Esaminiamo quello che succede, passo per passo.

Casting (2)

Il casting esplicito avviene quando:

```
Newtype nt; Oldtype ot; nt = (Newtype) ot;
```

Regole per il compilatore:

	`Oldtype` è una classe non-`final`	`Oldtype` è una classe `final`	`Oldtype` è una interfaccia	`Oldtype` è un array
`Newtype` è una classe non-`final`	`Oldtype` deve estendere `Newtype` o viceversa	`Oldtype` deve estendere `Newtype`	Sempre OK	`Newtype` deve essere `Object`
`Newtype` è una classe `final`	`Newtype` deve estendere `Oldtype`	`Oldtype` e `Newtype` devono coincidere	`Newtype` deve impl. l'interfaccia o impl. `Serializable`	Errore di compilazione
`Newtype` è una interfaccia	Sempre OK	`Oldtype` deve impl. l'interfaccia `Newtype`	Sempre OK	Errore di compilazione
`Newtype` è un array	`Oldtype` deve essere `Object`	Errore di compilazione	Errore di compilazione	I tipi delle componenti dei due array devono essere riferimenti convertibili per cast

Casting (3)

Regola breve (per il compilatore, nei casi più comuni):

- Quando sia `Oldtype`, sia `Newtype` sono classi, una classe deve essere sottoclasse dell'altra.
- Quando sia `Oldtype`, sia `Newtype` sono array, entrambi devono contenere riferimenti (non primitivi), e deve essere legale eseguire un cast tra gli elementi di `Oldtype` e quelli di `Newtype`.
- È sempre legale il cast tra una interfaccia e una classe non-`final`.

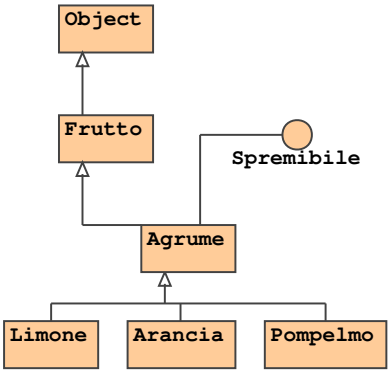
Casting (4)

Infine le ulteriori regole a cui deve obbedire un cast durante l'esecuzione (posto che sia sopravvissuto alla compilazione):

- Se `Newtype` è una classe, la classe originaria `Objtype` dell'oggetto deve essere `Newtype` oppure una sua sottoclasse.
- Se `Newtype` è una interfaccia, la classe originaria `Objtype` dell'oggetto deve implementare `Newtype`.

È tempo di fare alcuni esempi chiarificatori:

Esempi




```
Limone l, l1;
Agrume a;
Pompelmo p;
l = new Limone();
a = l;           // auto-conver.
l1 = (Limone)a;  // cast legale
p = (Pompelmo)a; // cast illeg.
                 // err. esecuz.
```

```
Limone l, l1;
Spremibile s;
l = new Limone();
s = l;          // OK
l1= s;          // err. compil.
l1= (Limone)s; // OK
```

Una interfaccia può essere auto-convertita solo ad `Object`.

```
Limone l[];
Spremibile s[];
Agrume a[];
l = new Limone[7];
s = l;           // OK
a = (Agrume[])s; // OK
```

È sempre lecito il cast di una interfaccia ad una classe non-`final` in compilazione. Qui il cast ha successo anche in esecuzione.

Questionario

D 1

Quale dei seguenti enunciati è corretto?

A. Solo i tipi primitivi sono convertiti automaticamente; per cambiare il tipo di un riferimento bisogna fare un cast.
B. Solo i riferimenti sono convertiti automaticamente; per cambiare il tipo di un primitivo bisogna fare un cast.
C. La promozione aritmetica di riferimenti richiede il cast esplicito.
D. Sia i tipi primitivi, sia i riferimenti possono essere convertiti sia automaticamente sia attraverso un cast.
E. Il cast dei tipi numerici richiede un controllo in esecuzione.

D 2

Quale dei seguenti enunciati è vero?

A. I riferimenti ad oggetti possono essere convertiti nelle assegnazioni e non nelle invocazioni di metodi.
B. I riferimenti ad oggetti possono essere convertiti nelle invocazioni di metodi e non nelle assegnazioni.
C. I riferimenti ad oggetti possono essere convertiti sia nelle assegnazioni, sia nelle invocazioni di metodi, ma le regole nei due casi sono diverse.
D. I riferimenti ad oggetti possono essere convertiti sia nelle assegnazioni, sia nelle invocazioni di metodi, e le regole nei due casi sono identiche.
E. I riferimenti ad oggetti non possono essere mei convertiti.

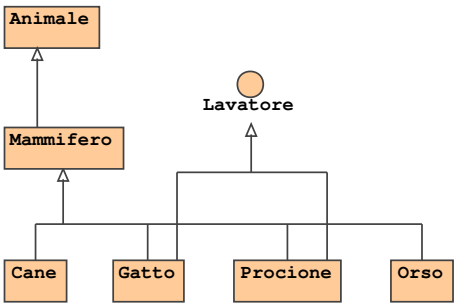
D 3

Quale linea del codice seguente non compila?

```
1. Object ob = new Object();
2. String stringarr[] = new String[50];
3. Float floater = new Float(3.14f);
4.
5. ob = stringarr;
6. ob = stringarr[5];
7. floater = ob;
8. ob = floater;
```

A. La linea 5.
B. La linea 6.
C. La linea 7.
D. La linea 8.

D 4



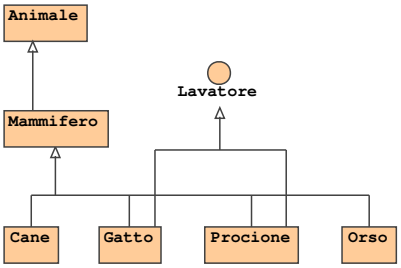
Si consideri il codice seguente:

```
1. Cane billy, fido;
2. Animale anim;
3.
4. billy = new Cane();
5. anim = billy;
6. fido = (Cane)anim;
```

Quale dei seguenti enunciati è vero?

A. La linea 5 non compila.
B. La linea 6 non compila.
C. Il codice è compilato correttamente ma viene lanciata una eccezione in esecuzione.
D. Il codice è compilato ed eseguito correttamente.
E. Il codice è compilato ed eseguito correttamente, ma il cast alla linea 6 è superfluo e può essere eliminato.

D 5



Si consideri il codice seguente:

```
1. Gatto fufi;
2. Lavatore lala;
3. Orso bubu;
4.
5. fufi = new Gatto();
6. lala = fufi;
7. bubu = (Orso)lala;
```

Quale dei seguenti enunciati è vero?

A. La linea 6 non compila poiché è richiesto un cast esplicito per convertire un `Gatto` ad un `Lavatore`.

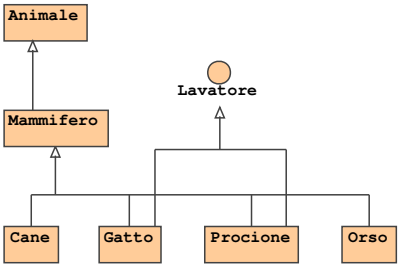
B. La linea 7 non compila poiché non si può fare il cast da una interfaccia ad una classe.

C. Il codice è compilato ed eseguito correttamente ma il cast alla linea 7 non è necessario.

D. Il codice è compilato ma, alla linea 7, lancia una eccezione in esecuzione poiché la conversione a runtime di una interfaccia in una classe è proibita.

E. Il codice è compilato ma, alla linea 7, lancia una eccezione poiché il tipo di `lala` in esecuzione non può essere convertito al tipo `Orso`.

D 6



Si consideri il codice seguente:

```
1. Procione rocco;
2. Orso bubu;
3. Lavatore lala;
4.
5. rocco = new Procione();
6. lala = rocco;
7. bubu = lala;
```

Quale dei seguenti enunciati è vero?

A. La linea 6 non compila: è richiesto un cast esplicito.

B. La linea 7 non compila: è richiesto un cast esplicito.

C. Il codice è compilato ed eseguito correttamente.

D. Il codice è compilato ma viene lanciata una eccezione alla linea 7, poiché in esecuzione la conversione da una interfaccia ad una classe è proibita.

E. Il codice è compilato ma viene lanciata una eccezione alla linea 7, poiché la classe dell'oggetto `lala` non può essere convertita al tipo `Orso`.