

Programmazione Orientata agli Oggetti

Interfacce e Polimorfismo
Upcasting e Downcasting

Contenuti

- Riferimenti tipati
- Java Interface
- Principio di sostituzione
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Contenuti

- Riferimenti tipati
- Java Interface
- Principio di sostituzione
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Riferimenti Tipati (1)

- In Java i riferimenti sono tipati, ovvero specificano il tipo dell'oggetto referenziato
- La definizione:
Strumento s;
afferma che **s** è un riferimento ad un oggetto di tipo **Strumento**
- Questo significa che attraverso **s** possiamo invocare i servizi del tipo **Strumento**
 - ovvero che è possibile chiedere di eseguire i metodi offerti dal tipo **Strumento** all'oggetto referenziato da **s**

Riferimenti Tipati (2)

- Consideriamo la seguente classe **Musicista**

```
public class Musicista {  
    private String nome;  
  
    public Musicista(String nome) {  
        this.nome = nome;  
    }  
  
    public void suona(Strumento s) {  
        s.produciSuono();  
    }  
}
```

Riferimenti Tipati (3)

- Il metodo `suona (Strumento s)` prende come parametro un riferimento ad un oggetto il cui tipo è `Strumento`
- Nel corpo del metodo è possibile invocare su `s` tutti i metodi offerti dal tipo `Strumento`
 - intuiamo che `Strumento` offre il metodo `public void produciSuono()`

Costrutti Java per la Definizione di Nuovi Tipi

- Fino ad ora abbiamo visto un solo modo per definire nuovi tipi: la definizione di nuove classi (mediante il costrutto **class**)
- In Java (e in altri moderni linguaggi OO, come ad esempio C#, Scala) esistono molteplici costrutti per definire nuovi tipi
- È il costrutto **interface**

Contenuti

- Riferimenti tipati
- **Java Interface**
- Principio di sostituzione
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Java Interface (1)

- Possiamo dire che una **interface** specifica un tipo in termini dei servizi, ovvero dei metodi, che questi può offrire
- Una **interface** non specifica i dettagli implementativi dei vari servizi, specifica solamente in che modo i servizi possono essere invocati (nome, parametri, tipo restituito)
- In definitiva una **interface** consiste in una specifica delle signature (e dei tipi restituiti) dai metodi che il tipo può offrire

Java Interface (2)

- Esempio:

```
public interface Strumento {  
    public void produciSuono();  
}
```

- ✓ L'interface **Strumento** definisce il tipo di oggetti che possono offrire il metodo **produciSuono()**

Java Interface (3)

- Nelle interface specifichiamo solo le signature (e il tipo restituito) dei metodi che un tipo può offrire
- In una **interface** non c'è nessun dettaglio relativo alla implementazione
 - Niente variabili
 - Niente costruttori
 - Niente corpo dei metodi
- Le **interface** non si possono istanziare
- Ma una classe può implementare una (o più) interface
- Una classe che implementa una **interface** garantisce che le sue istanze rispettino il tipo specificato nella **interface**

Classi ed Interface (1)

```
public class Tamburo implements Strumento {  
    public void produciSuono() {  
        System.out.println("bum-bum-bum");  
    }  
}
```

- La parola chiave **implements** serve a specificare che la classe **Tamburo** implementa l'interfaccia **Strumento**
- Questo significa che gli oggetti **Tamburo** sono in grado di offrire i metodi del tipo **Strumento**

Classi ed Interface (2)

- Una classe che implementa una **interface** può avere altri metodi (oltre a quelli della **interface**) specifici della classe

```
public class Chitarra implements Strumento {  
    private int[] corde;  
  
    public Chitarra() {  
        corde = new int[6];  
    }  
  
    public void produciSuono() {  
        System.out.println("dlen-dlen-dlen");  
    }  
  
    public int accorda(int corda, int val) {  
        return corde[corda] += val;  
    }  
}
```

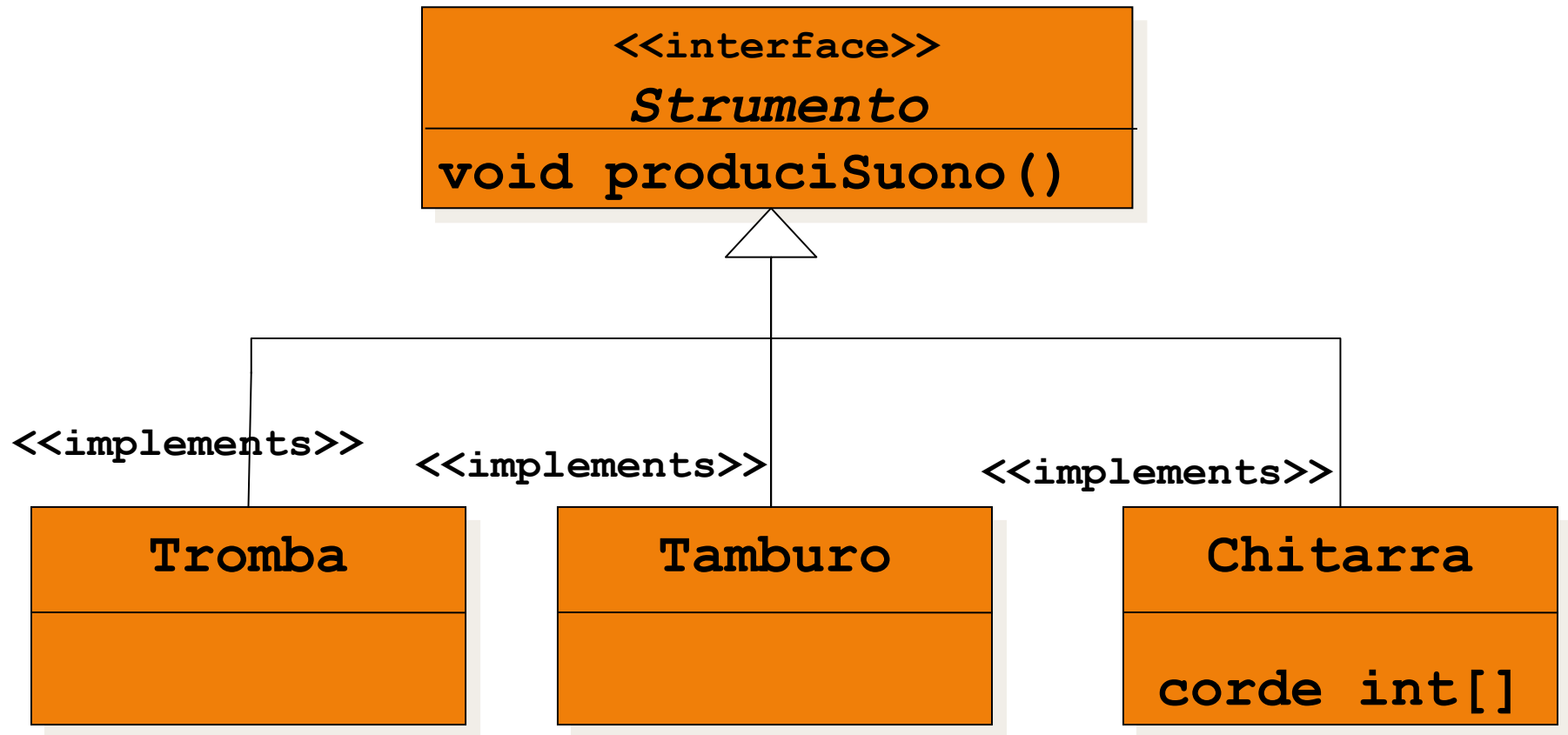
Esempio

```
public class Tamburo implements Strumento {  
    public void produciSuono() {  
        System.out.println("bum-bum-bum");  
    }  
}
```

```
public class Tromba implements Strumento {  
    public void produciSuono() {  
        System.out.println("pe-pe-re-pe-pe");  
    }  
}
```

```
public class Chitarra implements Strumento {  
    private int[] corde;  
    public Chitarra(){  
        corde = new int[6];  
    }  
    public void produciSuono() {  
        System.out.println("dlen-dlen-dlen");  
    }  
    public int accorda(int corda, int val) {  
        return corde[corda] += val;  
    }  
}
```

Diagramma delle Classi



Tipi, Sottotipi & Supertipi

- Abbiamo detto che una **interface** definisce un tipo
- Se la classe **C** implementa una **interface I** diciamo che:
 - **C** è un *sottotipo* di **I**
 - e che
 - **I** è un *supertipo* di **C**
- Ad esempio
 - **Tamburo** è un sottotipo di **Strumento**
 - **Strumento** è un supertipo di **Tamburo**

Contenuti

- Riferimenti tipati
- Java Interface
- **Principio di sostituzione**
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Principio di Sostituzione (1)

- In Java vale il *principio di sostituzione (di Liskov)*: un sottotipo può essere sempre usato in qualsiasi situazione in cui ci si aspetta un suo supertipo
- Rivediamo il metodo `suona (Strumento s)` della classe **Musicista**:

```
public void suona (Strumento s) {  
    s.produciSuono ();  
}
```

- Se invochiamo il metodo `suona (Strumento s)`, per il principio di sostituzione, possiamo passargli anche un riferimento ad un oggetto istanza di una qualunque classe che implementi l'interfaccia **Strumento**

Principio di Sostituzione (2)

- Esempio:

```
public static void main(String[] args){  
    Chitarra c = new Chitarra();  
    Strumento t = new Tamburo();  
    Musicista ludovico = new Musicista("Ludovico");  
    ludovico.suona(c);  
    ludovico.suona(t);  
}
```

- Nelle chiamate al metodo **suona(Strumento s)** abbiamo usato un riferimento ad un oggetto **Chitarra** (e poi un riferimento ad un oggetto **Tamburo**) al posto di un riferimento a **Strumento**

Principio di Sostituzione (3)

- Per il principio di sostituzione, un riferimento ad un sottotipo può essere assegnato ad un riferimento ad un suo supertipo
- Esempio:

```
Strumento s;
```

```
Chitarra c;
```

```
c = new Chitarra();
```

```
s = c;
```

Principio di Sostituzione (4)

- Commentiamo le precedenti istruzioni

Strumento s;

- Abbiamo definito una variabile **s**: contiene un riferimento ad un oggetto che rispetta il tipo **Strumento**

Chitarra c;

- Abbiamo definito una variabile **c**: contiene un riferimento ad un oggetto che rispetta il tipo **Chitarra**

c = new Chitarra();

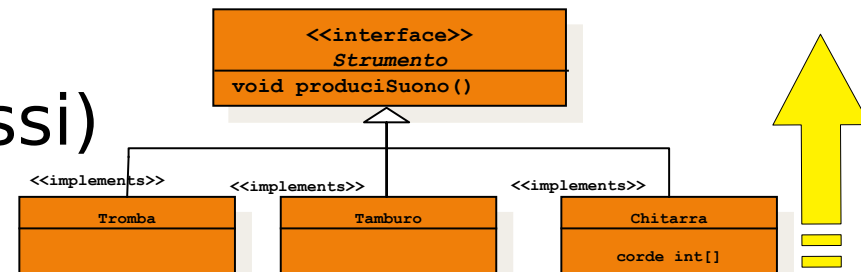
- Abbiamo creato un oggetto **Chitarra** e ne abbiamo assegnato il riferimento alla variabile **c**

s = c;

- Abbiamo assegnato il riferimento all'oggetto **c** alla variabile **s**
- È tutto lecito perché l'oggetto **c** è istanza della classe **Chitarra** che implementa il supertipo **Strumento**

Upcasting

- La promozione da un tipo ad un suo supertipo viene chiamata *upcasting*
- *upcasting* : un riferimento ad un oggetto è “promosso” in un riferimento ad un suo supertipo
- Il termine (“Up”=“verso l’alto”) è tradizionalmente legato al modo in cui vengono espresse graficamente le dipendenze supertipo/sottotipo (vedi diagramma delle classi)



Contenuti

- Riferimenti tipati
- Java Interface
- Principio di sostituzione
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Polimorfismo e Late Binding (1)

- Consideriamo la classe `Musicista`

```
public class Musicista {  
    private String nome;  
  
    public Musicista(String nome) {  
        this.nome = nome;  
    }  
  
    public void suona(Strumento s) {  
        s.produciSuono();  
    }  
}
```


Polimorfismo e Late Binding (2)

- Cosa succede a tempo di esecuzione, quando al parametro `s` è legato un oggetto?
- Sappiamo che il metodo `produciSuono()` viene invocato da un oggetto la cui classe implementa l'interfaccia **Strumento**
- Ma il codice da eseguire non è noto finché non siamo a tempo di esecuzione
- Il collegamento tra segnatura e corpo del codice da eseguire per `produciSuono()` viene stabilito solo a tempo di esecuzione (*late binding*)
- C'è un comportamento *polimorfo* del parametro formale **Strumento s**
 - può assumere forme/comportamenti diversi: tutti quelli dei suoi sottotipi

Contenuti

- Riferimenti tipati
- Java Interface
- Principio di sostituzione
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Tipo Statico e Tipo Dinamico

- Consideriamo la seguente istruzione:

```
Strumento s = new Chitarra();
```

- È lecita, per il principio di sostituzione
- Qual è il tipo della variabile `s`?
- Dobbiamo distinguere tra
 - Tipo statico
 - Tipo dinamico

Tipo Statico

- Il tipo statico è quello che viene usato nella dichiarazione della variabile
- Ad esempio, nella istruzione:
`Strumento s = new Chitarra();`
il tipo statico di `s` è `Strumento`
- Il tipo statico è determinato a tempo di compilazione
- Il compilatore permette di applicare i metodi del tipo statico (ovvero verifica che su una variabile siano invocati i metodi del suo tipo statico)
- Nel nostro esempio possiamo invocare su `s` solo i metodi di `Strumento`

```
Strumento s = new Chitarra();  
s.produciSuono(); // CORRETTO  
s.accorda(2,1);  // ERRATO: il tipo Strumento non  
                // possiede il metodo accorda(int, int)
```

Tipo Dinamico

- Il tipo dinamico è quello dell'oggetto realmente istanziato e quindi referenziato in memoria

- Ad esempio, nella istruzione:

```
Strumento s = new Chitarra();
```

il tipo dinamico di **s** è **Chitarra**

- Il tipo dinamico stabilisce quale sarà l'implementazione usata

- Nel nostro esempio:

```
Strumento s = new Chitarra();
```

```
s.produciSuono();
```

- A tempo di esecuzione il codice del metodo `produciSuono()` che viene usato è quello definito nella classe `chitarra`

Tipo Statico e Tipo Dinamico

- Capire la differenza tra tipo statico e tipo dinamico è fondamentale
- Il tipo statico viene assegnato dal compilatore e determina l'insieme dei metodi che possono essere invocati
- Il tipo dinamico interviene a tempo di esecuzione e determina l'implementazione che viene eseguita

Tipo Statico vs Tipo Dinamico (1)

Qual'è il tipo di c?

```
Chitarra c = new Chitarra();
```

Tipo Statico

Tipo Dinamico

Qual'è il tipo di s?

```
Strumento s = new Chitarra();
```

Tipo Statico

Tipo Dinamico

Tipo Statico vs Tipo Dinamico (2)

- Il tipo dichiarato di una variabile è il suo *tipo statico*
- Il tipo dell'oggetto a cui una variabile si riferisce è il suo *tipo dinamico*
- Il compilatore si preoccupa di verificare violazioni del tipo statico

```
Strumento strumento = new Chitarra();  
strumento.accorda(2,1); // ERRORE a tempo di compilazione
```

- **accorda()** non è tra i metodi di **Strumento** (tipo statico della var. locale **strumento**)

Tipo Statico vs Tipo Dinamico (3)

- A tempo di esecuzione viene eseguito il metodo del tipo dinamico
 - d'altronde i metodi definiti nelle interfacce non possiedono implementazione se non quella delle classi che le implementano
- Nota che il compilatore non solo non conosce, ma neanche può prevedere, in generale, i tipi dinamici >>

Imprevedibilità dell'Esecuzione (1)

```
import java.util.Random;

public class OrchestraCausale {
    public static void main(String[] args){
        Strumento[] orchestra = new Strumento[10];
        Random r = new Random();

        for(int i=0; i<orchestra.length; i++) {
            int numeroAcaso = r.nextInt(3);
            if (numeroAcaso==0)
                orchestra[i] = new Chitarra();
            if (numeroAcaso==1)
                orchestra[i] = new Tamburo();
            if (numeroAcaso==2)
                orchestra[i] = new Tromba();
        }

        for(int i=0; i<orchestra.length; i++)
            orchestra[i].produciSuono();
    }
}
```

Imprevedibilità dell'Esecuzione (2)

- Nell'esempio precedente l'array è riempito casualmente a tempo di esecuzione: non sappiamo a priori quali strumenti vengono assegnati ai vari elementi dell'array
- A tempo di esecuzione, ogni elemento dell'array produce il suono corrispondente al tipo dinamico
- A tempo di compilazione, ogni elemento dell'array possiede tipo statico **Strumento**

Tipo Statico e Tipo Dinamico: Overloading (1)

- L'overloading dei metodi viene risolto dal compilatore, quindi staticamente
- In particolare:
 - se abbiamo un metodo sovraccarico il compilatore guarda il tipo statico dei parametri per decidere qual'è il metodo da invocare
- Vedi esercizio seguente

Tipo Statico e Tipo Dinamico: Overloading (2)

```
interface Edificio {  
    public int altezza();  
}
```

```
public class Palazzo implements Edificio {  
    private int altezza;  
    public Palazzo(int altezza) {this.altezza = altezza;}  
    public int altezza() {return this.altezza;}  
}
```

```
public class Coloratore {
```

```
    public void colora(Edificio e) {  
        System.out.println("Colorato Edificio");  
    }
```

```
    public void colora(Palazzo p) {  
        System.out.println("Colorato Palazzo");  
    }
```

```
    public static void main(String args[]) {  
        Palazzo p = new Palazzo(4);  
        Edificio e = new Palazzo(3);  
        Coloratore c = new Coloratore();
```

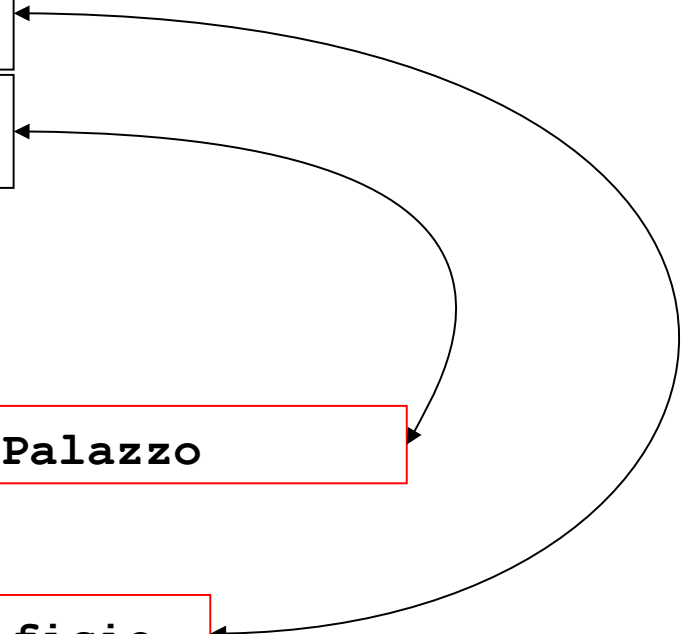
```
        c.colora(p);  
        c.colora(e);
```

```
    }
```

```
}
```

Tipo statico di p è Palazzo

Tipo statico di e è Edificio



Tipo Statico e Tipo Dinamico: Esempio

```
interface Veicolo {  
    public void func(Veicolo v);  
    public void func(Autotreno a);  
}  
  
public class Autotreno implements Veicolo {  
    public void func(Veicolo v) {  
        System.out.println("Autotreno.func(Veicolo) ");  
    }  
    public void func(Autotreno a) {  
        System.out.println("Autotreno.func(Autotreno) ");  
    }  
  
    public static void main(String args[]) {  
        Veicolo a = new Autotreno();  
        Autotreno b = new Autotreno();  
        a.func(b);  
  
        a.func(a);  
    }  
}
```

Tipo statico di b è
Autotreno

Tipo statico di a è Veicolo

Esercizi

- Fare le verifiche
 - L.java
 - Olimpiadi.java
 - Villa.java

Contenuti

- Riferimenti tipati
- Java Interface
- Principio di sostituzione
- Polimorfismo e late binding
- Tipo statico e tipo dinamico
- Interfacce come ruolo

Interface come *Ruolo* (1)

- Una classe può implementare più di una interface
- Potremmo dire che ciascuna interface implementata da una classe rappresenta uno specifico "ruolo" che la classe può assumere
- Ragionare sui ruoli (ed usare le potenzialità del polimorfismo) ci aiuta a produrre codice altamente riutilizzabile

Interface, Ruoli e Riuso (1)

- Consideriamo un problema noto che si presta naturalmente ad un comportamento polimorfo degli oggetti interessati: l'ordinamento
- Supponiamo di avere una classe che modella un "orario", espresso in ore e minuti
- Supponiamo di avere una collezione (per semplicità un array) di oggetti orario
- Supponiamo di voler ordinare questa collezione

Esempio: La Classe Orario

```
public class Orario {
    private int ore;
    private int minuti;

    public Orario(int ore, int minuti) {
        this.ore = ore;
        this.minuti = minuti;
    }

    public int getOre() {
        return this.ore;
    }

    public int getMinuti() {
        return this.minuti;
    }

    public boolean minoreDi(Orario o) {
        if (this.getOre() > o.getOre())
            return false;
        if (this.getOre() == o.getOre())
            return (this.getMinuti() < o.getMinuti());
        return true;
    }

    public String toString() {
        return this.getOre()+":"+this.getMinuti();
    }
}
```

Interface, Ruoli e Riuso (2)

- Per ordinare la collezione creiamo una opportuna classe che offre questa funzionalità attraverso il metodo `ordina(Orario[])`
- Scriviamo il codice
(usiamo un qualsiasi algoritmo di ordinamento, cfr corso Fondamenti, ad esempio il «selection sort»)
- vedi classe `OrdinatoreOrari`

La Classe OrdinatoriOrari

```
public class OrdinatoriOrari {  
  
    public static void ordina(Orario[] lista) {  
        int imin;  
        for (int ord=0; ord<lista.length-1; ord++) {  
            imin = ord;  
            for (int i=ord+1; i<lista.length; i++)  
                if (lista[i].minoreDi(lista[imin])) {  
                    Orario temp=lista[i];  
                    lista[i]=lista[imin];  
                    lista[imin]=temp;  
                }  
            }  
        }  
    }  
}
```

Interface, Ruoli e Riuso (3)

- Osserviamo bene il codice di **OrdinatoreOrari**
- Affinché gli oggetti dell'array possano essere ordinati, l'unica proprietà che questi oggetti devono avere è quella di possedere un metodo **minoreDi (Orario)**
- In altri termini l'ordinamento funziona su oggetti che sappiano interpretare il *ruolo* di «essere confrontati»
- Questo ruolo lo possiamo esplicitare in una opportuna interface

“Confrontabilità”, come Ruolo

- Creiamo l'interface **Comparable**: gli oggetti delle classi che la implementano sono in grado di essere confrontati tramite il metodo **minoreDi (Comparable)**

L'interface Comparabile

```
public interface Comparabile {  
    public boolean minoreDi (Comparabile c) ;  
}
```


Interface, Ruoli e Riuso (4)

- Possiamo ora generalizzare la nostra classe **Ordinatore** (e il relativo algoritmo di ordinamento) affinché funzioni su tutte le classi che sappiano interpretare il ruolo **Comparabile**

La Classe Ordinatori

```
public class Ordinatori {  
  
    public static void ordina(Comparabile[] lista) {  
        int imin;  
        for (int ord=0; ord<lista.length-1; ord++) {  
            imin = ord;  
            for (int i=ord+1; i<lista.length; i++)  
                if (lista[i].minoreDi(lista[imin])) {  
                    Comparabile temp=lista[i];  
                    lista[i]=lista[imin];  
                    lista[imin]=temp;  
                }  
            }  
        }  
    }  
}
```

La Classe Orario (rivisitata)

```
public class Orario implements Comparabile {  
    private int ore;  
    private int minuti;  
  
    public Orario(int ore, int minuti) {  
        this.ore = ore;  
        this.minuti = minuti;  
    }  
  
    public int getOre() {  
        return this.ore;  
    }  
  
    public int getMinuti() {  
        return this.minuti;  
    }  
}
```

```
public boolean minoreDi(Comparabile c) {  
    Orario o;  
    o = (Orario)c;  
    if (this.getOre() > o.getOre())  
        return false;  
    if (this.getOre() == o.getOre())  
        return (this.getMinuti() < o.getMinuti());  
    return true;  
}
```

Interface, Ruoli e Riutilizzo (5)

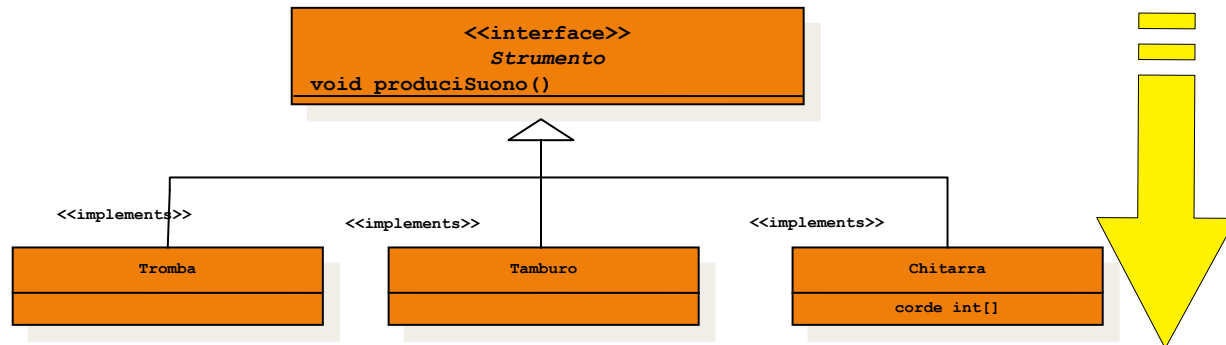
- Per rispettare l'interface **Comparabile** il metodo **minoreDi ()** deve prendere come parametro un oggetto **Comparabile**

```
public boolean minoreDi (Comparabile c)
```

- Quando però scriviamo il codice, dobbiamo poter usare i metodi specifici della classe **Orario** (altrimenti non potremmo implementare il metodo!)
- Il compilatore non ce lo permette: il tipo statico del parametro è **Comparabile**

Downcasting (1)

- È necessaria allora una “forzatura” sul tipo del parametro
- In particolare forziamo l'utilizzo (a tempo statico ed anche dinamico>>) del sottotipo
- Questa operazione viene chiamata *downcasting* (in opposizione all'*upcasting*)



Downcasting (2)

- Quando si forza il downcasting:
 - Il compilatore forza il tipo statico
 - La macchina virtuale effettua un controllo a tempo dinamico per verificare che l'operazione sia lecita e possibile
 - Ovvero verifica che l'oggetto appartenga al sottotipo a cui si sta forzando
- In caso contrario il programma abortisce sollevando una eccezione di tipo `java.lang.ClassCastException`

Esercizio

- 1) Scrivere con JUnit test-case *minimali* per confermare il corretto funzionamento del metodo `minoreDi()` come implementato nella classe `Orario`
- 2) Scrivere con JUnit un test-case per:
 - definire e creare un array di 5 oggetti `Orario`
 - creare 5 oggetti orario, che rappresentino i seguenti orari: 12:30, 21:40, 9:20, 4:00, 1:35
 - mettere i riferimenti ai 5 oggetti creati negli elementi dell'array
 - Ordinare l'array
 - Verificare che sia correttamente ordinato

Esercizio

- Scrivere una classe **Studente**, che contenga i campi nome (una stringa), età (un intero), un costruttore con due parametri, e i metodi accessori
- La classe **Studente** deve implementare l'interfaccia **Comparabile**, descritta in precedenza (vedi codice di **Orario**)
- Scrivere un metodo che crea un array di oggetti **Studente** e lo ordina (per età) usando il metodo **Ordinatore.ordina()**
- Scrivere con JUnit una classe di test per verificare che l'array sia effettivamente ordinato, dopo l'invocazione del metodo **Ordinatore.ordina()**

Esercizio

- Introdurre nell'interfaccia **Comparabile** un nuovo metodo
`int compara(Comparabile c)`
che restituisce un valore negativo, pari a 0, positivo, se l'oggetto su cui è chiamato il metodo è rispettivamente minore, uguale, maggiore del valore del parametro
- Nella classe **Ordinatore**, scrivere il codice del metodo:
`public static int
 ricercaBinaria(Comparabile[] v, Comparabile cercato)`
che implementa l'algoritmo di ricerca binaria (cfr. corso di Fondamenti di Informatica); questo metodo restituisce un intero il cui valore corrisponde alla posizione dell'elemento **cercato** nell'array **v** oppure a -1 se tale elemento non è presente
- Scrivere, utilizzando JUnit, una classe di test per verificare il corretto funzionamento del metodo