Programmazione orientata agli oggetti

Tipo statico e tipo dinamico

Approfondimenti Interface

Classe Object

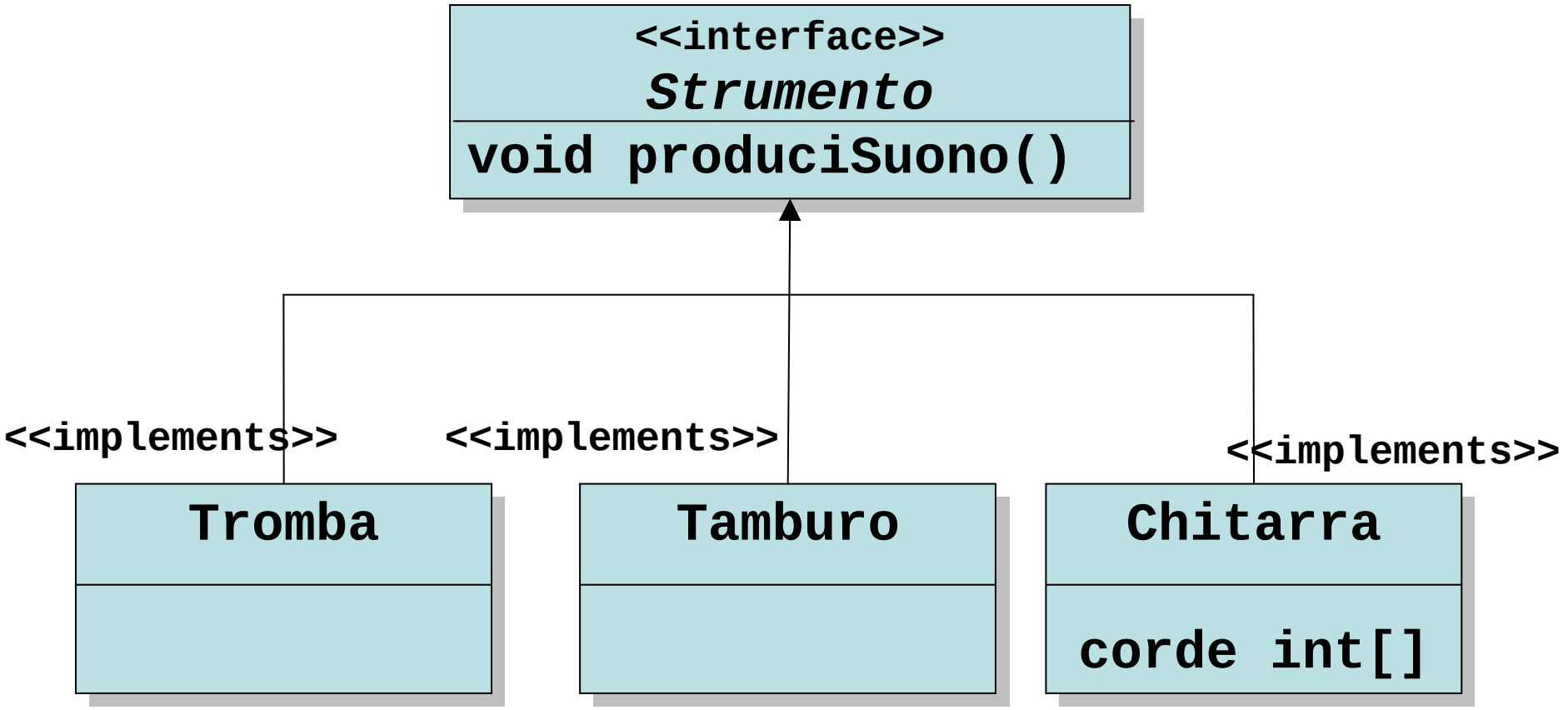
Esempio

```
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`**
- E **`Strumento`** è un supertipo di **`Tamburo`**

Principio di sostituzione

- In Java vale il *principio di sostituzione*:
  un sottotipo può essere sempre usato in qualsiasi situazione in cui ci si aspetta un suo supertipo
- Immaginiamo un metodo **`suona(Strumento s)`** di una 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 oggetto istanza di una qualunque classe che implementi l'interfaccia **`Strumento`**

Principio di sostituzione

- Esempio:

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

- 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

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

```
Strumento strumento;
Chitarra chitarra;
chitarra = new Chitarra();
strumento = chitarra;
```

Principio di sostituzione

- Commentiamo le precedenti istruzioni

  `Strumento strumento;`

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

  `Chitarra chitarra;`

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

  `chitarra = new Chitarra();`

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

  `strumento = chitarra;`

  - Abbiamo assegnato il riferimento all'oggetto **`chitarra`** alla variabile **`strumento`**

- È tutto lecito perché l'oggetto **`chitarra`** è istanza della classe **`Chitarra`**, che implementa il supertipo **`Strumento`**

Upcasting

- La promozione da un tipo ad un suo supertipo viene chiamata *upcasting*
- *upcasting* : prendere un riferimento ad un oggetto e promuoverlo in un riferimento ad un suo supertipo

- NOTA: il termine è legato al modo con cui tradizionalmente vengono espresse graficamente le dipendenze supertipo-sottotipo

  (vedi diagramma delle classi)

Riferimenti tipati

- 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

- 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 S**`trumento`**
- 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

Tipo statico e tipo dinamico

- Consideriamo la seguente istruzione:

  `Strumento strumento = new Chitarra();`
- È equivalente a
  - **`Strumento strumento; //dichiarazione`**
  - **`strumento = new Chitarra(); //istanziazione`**
- È lecita, per il principio di sostituzione
- Qual è il tipo della variabile **`strumento`**?
- 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 invocare 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 ha
                 //      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 al momento dell’istanziamento tramite l’operatore **new**, e determina l'implementazione del metodo che viene eseguita

Tipi statici e tipi dinamici

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

Tipi statici e tipi dinamici

- 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 tempo di compilazione
```

- **`accorda()`** non è tra i metodi di **`Strumento`** (tipo statico di **`strumento`**)

Tipi statici e tipi dinamici

- A tempo di esecuzione viene eseguito il metodo del tipo dinamico
  - In particolare, per quanto concerne le interfacce, i metodi ivi definiti non possiedono implementazione, se non quella delle classi che le implementano.

- Si noti che, in generale, il compilatore non solo non conosce, ma neanche può prevedere i tipi dinamici >>

Polimorfismo

```
import java.util.Random;

public class TestPolimorfismo {
   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();
   }
}
```

Tipo statico e dinamico; overloading

- 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 dinamico; overloading

```
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**

Estendere una interface

- Talvolta può essere utile definire una nuova interface a partire da una interface esistente
- Questo significa definire una nuova interface che offre qualche servizio (metodo) **aggiuntivo** rispetto ad una interface nota

Estendere una interface

- Esempio: data l'interface **A**

```
public interface A {
   public void a1(int i);
   public String a2();
}
```

- Supponiamo di dover definire l'interface **B**, che offre gli stessi metodi di **A**, ed in più offre il metodo **`b1`**

Estendere una interface

- Potremmo definire la nuova interface in questo modo

```
public interface B {
    public void a1(int i);
    public String a2();
    public int b1();
}
```

- Ma le due interface non hanno nessuna relazione
- Di conseguenza non potremmo referenziare un oggetto **B** con un riferimento di tipo **A**, anche se concettualmente sembrerebbe sensato

```
A a = new ClasseCheImplementa_A();
B b = new ClasseCheImplementa_B();
a=b; // ERRORE
```

Estendere una interface

- Sarebbe utile e sensato riuscire a specificare che **B** è un sottotipo di **A**
- In Java (e analogamente in altri linguaggi) questo è possibile definendo una interface come una *estensione* di un'altra interface
- Relativamente al nostro esempio potremmo scrivere

```
public interface B extends A {
   public int b1();
}
```

Estendere una interface

- In questo modo stiamo definendo una nuova interface (**B**) a partire da una interface esistente (**A**)
- In particolare stiamo dicendo che:
  - **B** è un sottotipo di **A**
  - **B** offre tutti i metodi di **A** più il metodo **b1**
- Quindi vale il principio di sostituzione
- In questo caso le istruzioni:

```
A a = new ClasseCheImplementa_A();
B b = new ClasseCheImplementa_B();
a = b; // OK B è un sottotipo di A
```

sono corrette

Estendere una interface

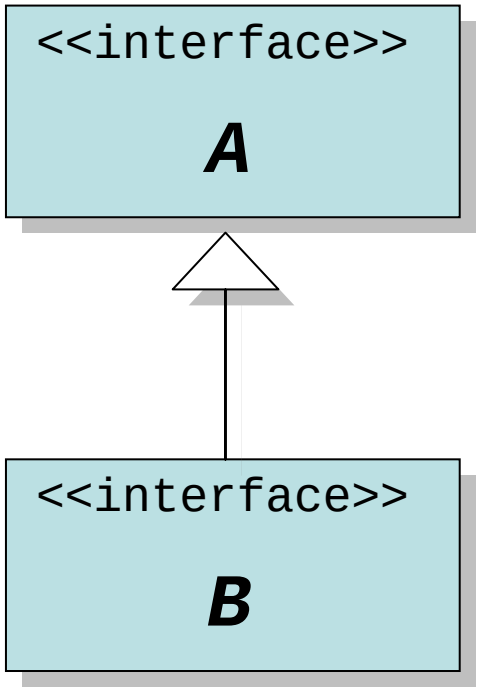
- Non pensiamo che l'estensione sia una scorciatoria per non ripetere la scrittura di metodi
- E' invece un sofisticato meccanismo per definire sottotipi
- Usare correttamente l'estensione delle interface (ma non solo) richiede esperienza
  - Il legame tra le due interface è forte e deve quindi essere giustificato

Estendere una interface

- In questo corso non useremo mai l'estensione delle interface
- Però dobbiamo essere in grado di capirne la semantica, perché useremo interface definite per estensione che sono presenti nelle API di Java

Estendere una interface

- Rappresentazione diagrammatica



Meccanismi per la creazione di tipi

- Attraverso l'estensione delle interfacce è possibile definire nuovi tipi a partire da tipi già esistenti
- Riassumiamo tutti i meccanismi per introdurre nuovi tipi in java
- Sfruttiamo l'occasione per introdurre la classe **`Object`**, che al contrario conviene comprendere prima del prossimo argomento, le collezioni

Creazione di tipi ex-novo

- Le interfacce
  - permettono di definire nuovi tipi senza definire l'implementazione dei metodi che formano la specifica di tipo
- Le classi
  - permettono di definire nuovi tipi ma richiedono l'implementazione di tutti i metodi che formano la specifica di tipo
- Le classi astratte
  - strumento intermedio che permette di lasciare qualche metodo "astratto", ovvero senza implementazione, pur consentendo l'implementazione di altri

Creazione di nuovi tipi definiti sulla base di tipi preesistenti

- Estensione di interfacce
  - nuove interfacce definite come estensione di altre, l'insieme dei metodi delle interfaccia estesa comprende quelli della classe base più altri di nuova definizione. Nessun metodo possiede implementazione nelle due interfacce
- Estensione di classi
  - nuove classi definite come estensioni di altre, l'insieme dei metodi della classe estesa comprende quelli della classe base non privati, più eventuali altri di nuova definizione. I metodi ereditano anche l'implementazione, che al limite può essere sovrascritta (*override*) nella classe estesa

Estensione: esempio

```
public class Persona {
	private String nome;

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

	public String getNome() {
		return this.nome;
	}
}
```

```
public class Studente extends Persona {
	private String matricola

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

	public String getMatricola() {
		return this.matricola;
	}
}
```

Estensione

- Vale il principio di sostituzione: il sottotipo può essere usato al posto di un suo supertipo
- Le variabili di istanza e i metodi vengono ereditati

```
public class ProvaPersona {

	public static void main(String[] args) {
		Persona p = new Studente();
		p.setNome("Anna");
		System.out.println(p.getNome());
		Studente s = new Studente();
		s.setNome("Antonio");
	}
}
```

La Classe `Object`

- In Java tutte le classi estendono automaticamente la classe **`Object`**
- Ad es.:

```
public class String extends Object {…}
```

- Attenzione: il compilatore considera le classi estensioni di **`Object`** anche quando non viene esplicitamente dichiarato nella sua definizione.

  Per il compilatore risulta equivalente:

```
public class String {…}
```

La gerarchia delle classi Java: `Object`

- In Java tutte le classi estendono automaticamente la classe **`Object`**
- E' una classe predefinita, che viene automaticamente estesa da ogni nuova classe (direttamente o indirettamente)
- La classe **`Object`** ha un insieme di metodi, che sono ereditati e possono essere ridefiniti da ogni nuova classe
- Tra questi metodi ce ne sono alcuni già noti ed altri che lo saranno presto!

`Object`: alcuni metodi (I)

`protected Object clone()`

- crea una nuova istanza della classe dell'oggetto, e ne inizializza i campi per assegnamento.

`protected void finalize()`

- viene invocato dal *Garbage Collector* quando non ci sono più riferimenti dell'oggetto.

`public boolean equals(Object obj)`

- verifica se i riferimenti dell'oggetto implicito e dell'oggetto obj sono uguali

`public Class getClass()`

- restituisce l'oggetto *Class* associato all'oggetto *this*.

`public int hashCode()`

- restituisce l'indirizzo interno dell'oggetto.

`public String toString()`

- restituisce una descrizione testuale dell'oggetto (nome classe ed indirizzo di memoria)

`Object`: alcuni metodi (II)

- Di tutti questi metodi le nostre classi ereditano l'implementazione, oltre che la segnatura
- Le nostre classi possono ridefinirne l'implementazione (rispettandone rigidamente la segnatura)

`Object.toString()`

- Questo è il motivo per il quale non è strettamente necessario definire il metodo **`toString()`** dentro le classi di nostra definizione per poterlo usare
- Se non lo definiamo, verrà comunque ereditata la definizione del metodo **`toString()`** propria della classe **`java.lang.Object`**
- Questa si preoccupa di stampare un messaggio testuale che rappresenta l'indirizzo in memoria dell'oggetto sul quale viene invocato
- Di solito risulta poco esplicativo, e perciò conviene ridefinirlo sulla base delle particolarità della classe definita

Object.toString(): esempio

```
public class Persona {
   private String codiceFiscale;
   private String nome;

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

   public static void main(String[] args) {
       Persona p = new Persona();
       p.setNome("Antonio");
       p.setCodiceFiscale("A34");
       System.out.println(p.toString());
   }
}
```

Stampa (qualcosa simile a): **Persona@10b62c9**

Il metodo `toString()` viene ereditato da `Object`: vale l'implementazione di `Object`

Object.toString(): esempio

```
public class Persona {
   private String codiceFiscale;
   private String nome;

   public void setNome(String nome){
        this.nome = nome;
   }
   public void setCodiceFiscale(String codiceFiscale){
        this.codiceFiscale=codiceFiscale;
   }
   public String toString() {
        return “Mi chiamo “ + this.nome+ ”. Codice fiscale: ” + codiceFiscale;
   }

   public static void main(String[] args) {
        Persona p = new Persona();
        p.setNome("Antonio");
        p.setCodiceFiscale(“A34")
        System.out.println(p.toString());
   }
}
```

Stampa: **`Mi chiamo Antonio. Codice fiscale: A34`**

Il metodo `toString()` è stato riscritto:
vale l'implementazione riscritta

Object: il metodo *equals(Object obj)*

- Viene utilizzato per stabilire se l'oggetto this e l'oggetto riferito dal parametro formale obj sono uguali.
- L'implementazione di default verifica che entrambi riferiscano il medesimo oggetto in memoria
  - Restituisce *true* se e solo se *this*==*obj*;
- In generale, ogni classe dovrebbe ridefinire il metodo equals ereditato da Object

Object: il metodo *equals(Object obj)*

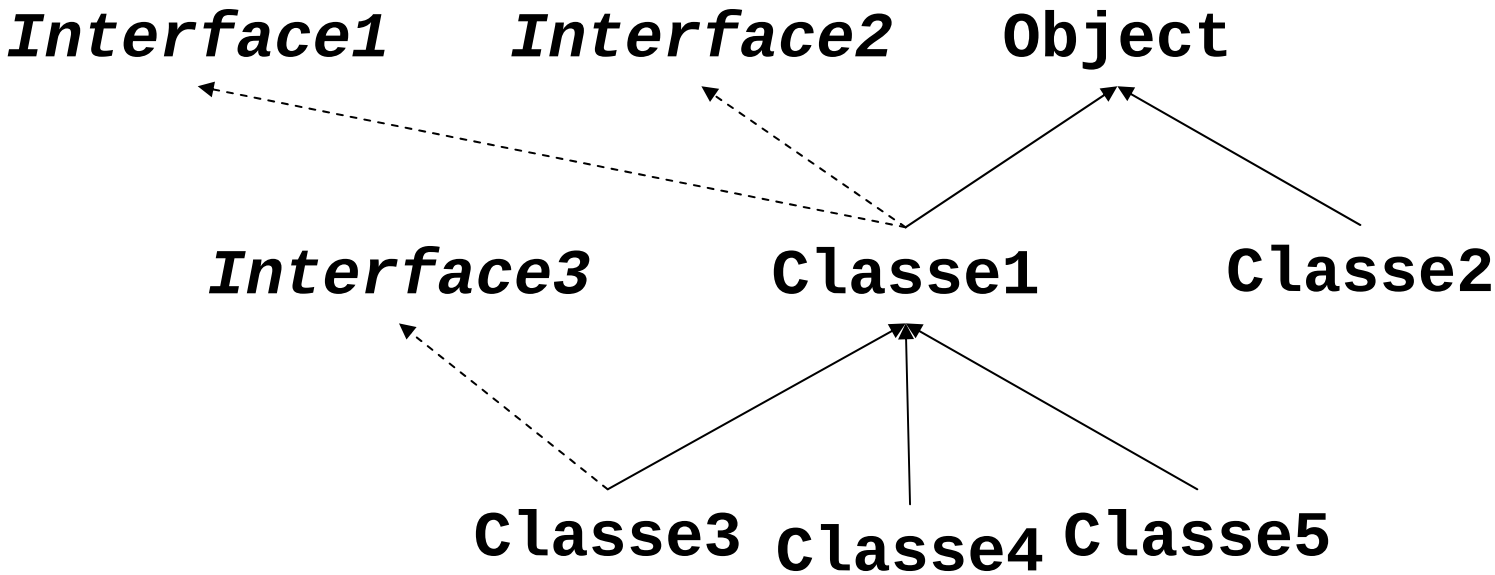
```
public class Persona {
   private String codiceFiscale;
   private String nome;
   public void setNome(String nome){
        this.nome = nome;
   }
   public void setCodiceFiscale(String codiceFiscale){
        this.codiceFiscale=codiceFiscale;
   }
   public String toString() {
        return “Mi chiamo “ + this.nome+ ”. Codice fiscale: ” + codiceFiscale;
   }
   public boolean equals(Object obj){
        if ((obj!=null) && (obj instanceof Persona)){
                Persona p = (Persona)obj; //downcasting
                return  this.codiceFiscale.equals(p.codiceFiscale);
        }
        else
                return false;
   }
   public static void main(String[] args) {
        Persona p = new Persona();
        p.setNome("Antonio");
        p.setCodiceFiscale(“A34")
        System.out.println(p.toString());
   }
}
```

Gerarchie di classi

- In Java ogni classe estende sempre una ed una sola classe
- Tranne **`Object`**, che è la radice predefinita delle classi
- Non ci può essere ereditarietà multipla
- Ma una classe può essere estesa da molte classi
- Ribadiamo che una classe può implementare zero, ma anche molteplici interfacce

La gerarchia delle classi in Java

- Un'unica radice: **`Object`**
- Ogni classe* ha una e una sola superclasse
- Ogni classe* può avere zero o più sottoclassi



* con l'eccezione di **`Object`**

La gerarchia delle classi in Java

- Si noti che il meccanismo di estensione delle classi rende possibili scenari come quello di seguito rappresentato