

Ereditarietà e Polimorfismo

Luca Grilli

Ereditarietà

- una delle caratteristiche distintive del paradigma di programmazione orientata agli oggetti è l'**ereditarietà**
 - l'ereditarietà rende possibile la definizione di nuove classi mediante l'aggiunta e/o la specializzazione di funzionalità a classi già esistenti
 - il linguaggio Java utilizza tre meccanismi di ereditarietà
 - l'**estensione di classi**
 - le **classi astratte**
 - le **interfacce**
- si vogliono ora descrivere questi tre meccanismi

Estensione di classi

- **l'estensione di classi** è il meccanismo che permette di
 - definire una classe a partire da un'altra classe (preesistente)
 - mediante l'aggiunta e/o la specializzazione di funzionalità (variabili e metodi)
- la classe preesistente si chiama **classe base** o **super-classe**
- la classe che viene definita si chiama **classe estesa** o **classe derivata** o **sotto-classe**

Estensione di classi

- tutte le operazioni definite dalla classe base sono implicitamente definite anche nella classe estesa
 - la classe estesa può definire delle nuove funzionalità per i propri oggetti
 - la classe estesa può ridefinire le funzionalità definite nella classe base
- ogni istanza della classe estesa può essere considerata anche una istanza della classe base

Esempio: la classe **Persona**

- La classe **Persona** modella persone reali nel seguente modo:
 - un oggetto **Persona** rappresenta una persona
 - le proprietà di una **Persona** sono il nome, il cognome, e il codice fiscale
 - è possibile costruire un oggetto **Persona** specificando il suo nome, cognome e codice fiscale
 - a una **Persona** è possibile chiedere il suo nome, cognome e codice fiscale
 - a una **Persona** è possibile chiedere una sua descrizione

La classe Persona

```
class Persona {  
  
    /* Proprietà di una persona. */  
    private String nome;  
    private String cognome;  
    private String codiceFiscale;  
  
    /* Crea una persona specificando il nome, il cognome e il  
       codice fiscale. */  
    public Persona(String nome, String cognome,  
                   String codiceFiscale) {  
        this.nome = nome;  
        this.cognome = cognome;  
        this.codiceFiscale = codiceFiscale;  
    }  
  
    /* Restituisce il nome. */  
    public String getNome() {  
        return this.nome;  
    }  
}
```

... continua dietro ...

La classe **Persona**

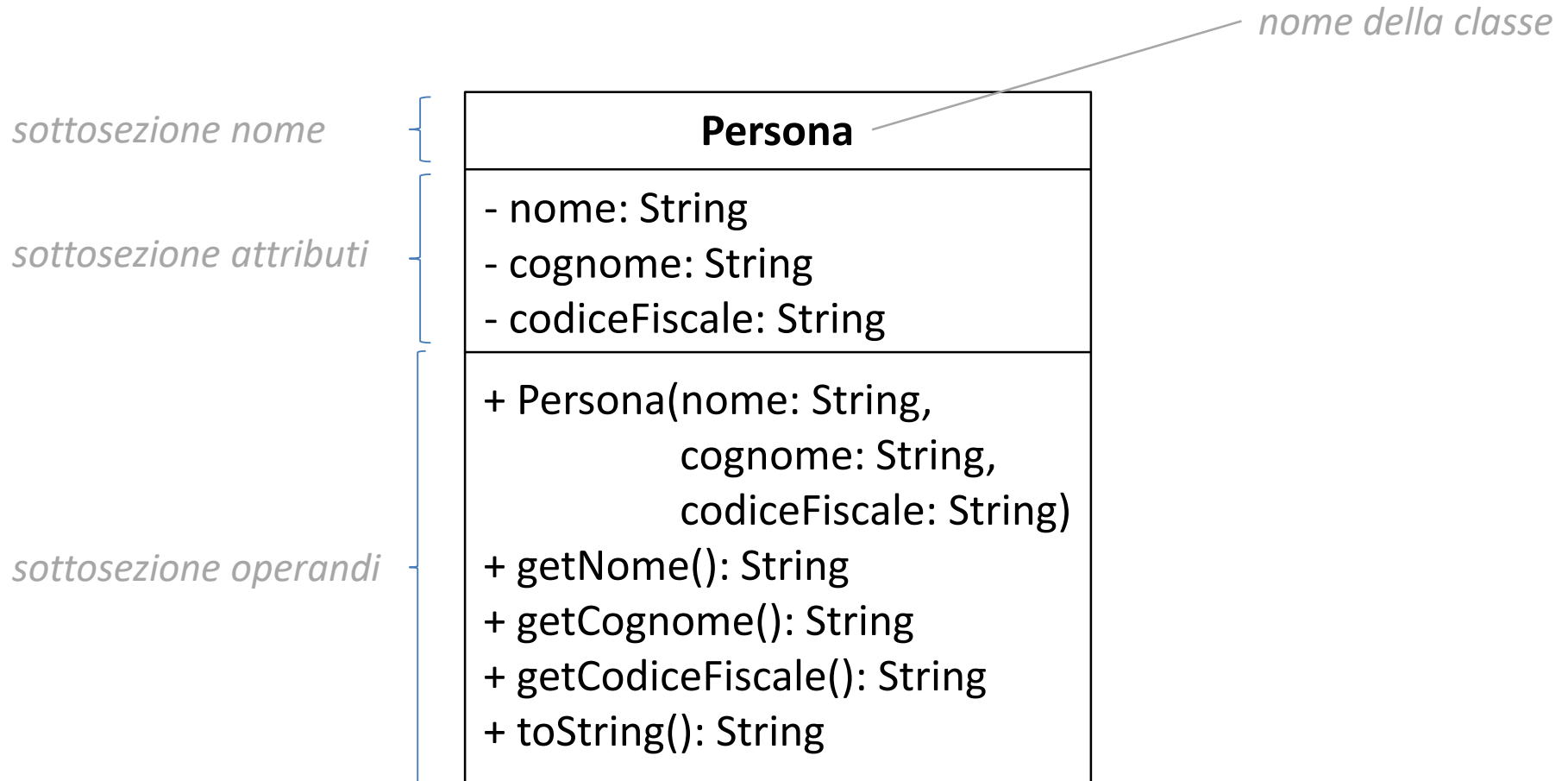
```
/* Restituisce il cognome. */
public String getCognome() {
    return this.cognome;
}

/* Restituisce il codice fiscale. */
public String getCodiceFiscale() {
    return this.codiceFiscale;
}

/* Restituisce una descrizione della persona. */
public String toString() {
    return "Mi chiamo " + this.nome + " " + this.cognome +
        " codice fiscale " + this.codiceFiscale;
}

} // end class
```

Classe Persona – diagramma UML



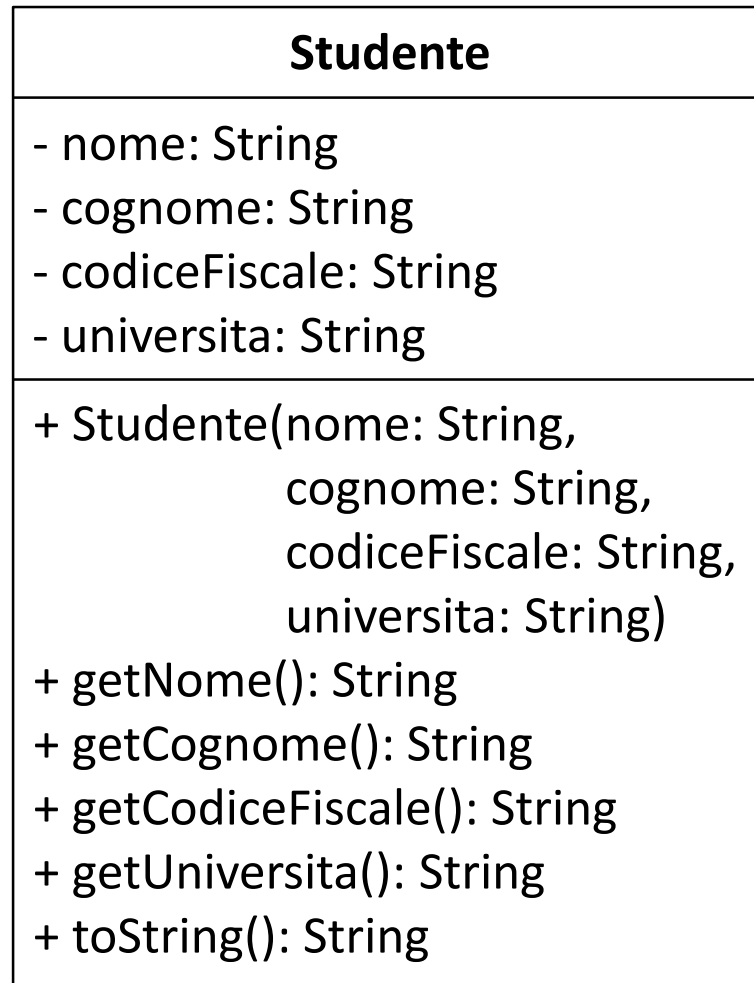
Esempio: la classe **Studente**

- si supponga ora di voler definire la seguente classe **Studente**
 - un oggetto **Studente** rappresenta uno studente universitario
 - le proprietà di uno **Studente** sono il suo nome, il cognome, il codice fiscale e il nome dell'università in cui studia
 - è possibile costruire un oggetto **Studente** specificando il suo nome, cognome, codice fiscale e il nome dell'università

Esempio: la classe **Studente**

- a uno **Studente** è possibile chiedere il suo nome, cognome, codice fiscale e il nome dell'università
- a uno **Studente** è possibile chiedere la sua descrizione

Classe Studente – diagramma UML



Persona vs Studente

Persona
<ul style="list-style-type: none">- nome: String- cognome: String- codiceFiscale: String
<ul style="list-style-type: none">+ Persona(nome: String, cognome: String, codiceFiscale: String)+ getNome(): String+ getCognome(): String+ getCodiceFiscale(): String+ toString(): String

Studente
<ul style="list-style-type: none">- nome: String- cognome: String- codiceFiscale: String- universita: String
<ul style="list-style-type: none">+ Studente(nome: String, cognome: String, codiceFiscale: String, universita: String)+ getNome(): String+ getCognome(): String+ getCodiceFiscale(): String+ getUniversita(): String+ toString(): String

Da **Persona** a **Studente**

- la classe **Studente** estende e specializza il comportamento della classe **Persona**
- la classe **Studente** modella oggetti che sono casi particolari della classe **Persona**
 - un oggetto **Studente** sa eseguire tutte le operazioni degli oggetti **Persona**
 - un oggetto **Studente** sa eseguire anche delle ulteriori operazioni, che gli oggetti **Persona** non sanno eseguire

Da **Persona** a **Studente**

- definiamo allora la classe **Studente** estendendo la classe **Persona**
 - l'estensione di classi riduce il tempo di realizzazione, riutilizzando implicitamente il codice già scritto
 - l'estensione di classi presenta anche altri vantaggi molto più importanti, che saranno descritti nel seguito

Studente come estensione di Persona

```
class Studente extends Persona {  
  
    /* Proprietà di uno studente, ma non di una generica persona. */  
    private String universita;  
  
    /* Crea uno studente specificando nome, cognome, codice fiscale  
       e università. */  
    public Studente(String nome, String cognome, String  
                    codiceFiscale, String universita) {  
        super(nome, cognome, codiceFiscale);  
        this.universita = universita;  
    }  
  
    /* Restituisce l'università dello studente. */  
    public String getUniversita() {  
        return this.universita;  
    }  
}
```

invoca il costruttore
della super-classe
Persona

... continua dietro ...

Studente come estensione di Persona

```
/* Restituisce una descrizione dello studente. */
public String toString() {
    return "Mi chiamo " + this.getNome() + " " +
           this.getCognome() + " codice fiscale " +
           this.getCodiceFiscale() + ". " +
           "Studio a " + this.universita + ".";
}
} // end class
```

Nota: le proprietà e i metodi pubblici della classe **Persona** vengono automaticamente ereditati dalla classe **Studente**

Persona vs Studente

Persona
- nome: String - cognome: String - codiceFiscale: String
+ Persona(nome: String, cognome: String, codiceFiscale: String) + getNome(): String + getCognome(): String + getCodiceFiscale(): String + toString(): String



Studente
- universita : String
+ Studente(nome: String, cognome: String, codiceFiscale: String, universita: String) + getUniversita(): String + toString(): String

<u>marioRossi : Persona</u>
- nome = "Mario" - cognome = "Rossi" - codiceFiscale = "RSSMRA85T10A562S"
+ getNome(): String + getCognome(): String + getCodiceFiscale(): String + toString(): String

<u>giuliaVerdi : Studente</u>
- nome = "Giulia" - cognome = "Verdi" - codiceFiscale = "GLIVRD80P60G478R" - universita = "Perugia"
+ getNome(): String + getCognome(): String + getCodiceFiscale(): String + getUniversita(): String + toString(): String

Aspetti nell'estensione di classi

- La sintassi per definire una sotto-classe è:

```
class <ClasseEstesa> extends <ClasseBase>
```

- Variabili di istanza della classe estesa
 - la classe estesa possiede implicitamente tutte le variabili di istanza della classe base
 - la classe estesa può accedere alle variabili di istanza pubbliche della classe base, ma non può accedere alle variabili private

Costruttori della classe estesa

- I costruttori della classe derivata devono
 - invocare un costruttore della classe base (usando la parola chiave **super**) per inizializzare le variabili dichiarate nella classe base
 - se ciò non viene fatto, è automaticamente richiamato il costruttore con parametri nulli (se non presente viene generato un errore)
 - inizializzare le variabili di istanza dichiarate nella classe estesa

Metodi della classe estesa

- La classe derivata può definire dei nuovi metodi rispetto a quelli definiti nella classe base
 - ad esempio, `getUniversita()`
- La classe derivata eredita implicitamente tutti i metodi pubblici definiti nella classe base, tuttavia
 - può ridefinire i metodi della classe base se ne vuole modificare la definizione (**overriding** o **sovrascrittura**) — ad esempio, `toString()`

Metodi della classe estesa

- **Osservazione:** Se la classe estesa non ridefinisce un metodo della classe base, allora vuol dire che ne conferma implicitamente la definizione
 - ad esempio, `getNome()`
- Se la definizione di un metodo della classe base non è più adeguata per la classe estesa, allora è necessario effettuare l'overriding (sovrascrittura) del metodo

Ereditarietà e polimorfismo

- Il **polimorfismo** è un aspetto essenziale dell'estensione di classe e dell'ereditarietà in generale
- Il **polimorfismo** consente di
 - referenziare oggetti di un tipo **TipoEsteso** con variabili riferimento di un altro tipo **TipoBase**
 - a patto che **TipoEsteso** sia un sotto-tipo di **TipoBase**
 - cioè l'oggetto referenziato deve essere un'istanza di una classe che estende (anche indirettamente) la classe **TipoBase**

Ereditarietà e polimorfismo

- Ad esempio,
 - un oggetto di tipo **Studente** può essere referenziato da una variabile riferimento di tipo **Persona**
 - non vale il viceversa, si otterrebbe un errore in fase di compilazione

Vincoli sintattici del polimorfismo

- Il compilatore
 - accetta che una variabile riferimento di tipo **TipoBase** referenzi un oggetto di tipo **TipoEsteso**
 - ma non permette che siano invocati metodi propri della classe **TipoEsteso**, cioè metodi non definiti nella classe **TipoBase**
- Visti i precedenti vincoli sintattici, a cosa mi serve il polimorfismo?

Utilità del polimorfismo

- Il comportamento polimorfico è legato all'uso di metodi sovrascritti.
- Supponiamo che il metodo *metodoA()* sia
 - definito nella classe **TipoBase**, e
 - sia ridefinito (sovrascritto) nella classe **TipoEsteso**
- Supponiamo inoltre che
 - una variabile riferimento *tipoBase*, di tipo **TipoBase**, referenzi un oggetto di tipo **TipoEsteso**

Utilità del polimorfismo

- Il compilatore permette
 - l'invocazione del metodo *metodoA()* sull'oggetto di tipo **TipoEsteso** referenziato dalla variabile *tipoBase*
 - *metodoA()* è definito anche nella classe **TipoBase**
- Tuttavia, in fase di esecuzione,
 - la JVM esegue la versione di *metodoA()* sulla base dell'oggetto referenziato
 - cioè viene eseguito il *metodoA()* dell'oggetto **TipoEsteso**

Eredità e polimorfismo – esempio

```
Persona giulia = new Studente("Giulia", "Verdi",  
                                "GLIVRD80P60G478R", "Perugia");  
  
System.out.println(giulia.toString());  
// Mi chiamo Giulia Verdi codice fiscale  
// GLIVRD80P60G478R. Studio a Perugia.  
  
// e non: Mi chiamo Giulia Verdi codice fiscale  
// GLIVRD80P60G478R.  
  
System.out.println(giulia.getUniversita());  
// NO, ERRORE!
```

Polimorfismo e parametri

- In modo simile, si può passare ad un metodo che ha un parametro formale di tipo **TipoBase** un parametro attuale che è un oggetto di tipo **TipoEsteso**
- Esempio
 - definizione del metodo

```
public static void stampa(Persona p) {  
    System.out.println(p.toString());  
}
```

- invocazione del metodo

```
NomeClasse.stampa(new Studente("Giulia", "Verdi",  
                                "GLIVRD80P60G478R", "Perugia"));  
// Mi chiamo Giulia Verdi codice fiscale  
// GLIVRD80P60G478R. Studio a Perugia.
```

Conversione Esplicita

- Si consideri ancora il caso di una variabile *tipoBase* di tipo **TipoBase** che referencia un dato oggetto
 - se si è sicuri che l'oggetto referenziato è di tipo **TipoEsteso**, allora è possibile effettuare una conversione esplicita del riferimento (cast), per ottenere un riferimento di tipo **TipoEsteso**
 - la conversione permette di utilizzare il comportamento specifico della classe estesa
 - la conversione genera un errore se l'oggetto referenziato non è di tipo **TipoEsteso**

Conversione esplicita – esempio

```
Persona marioP;  
Studente marioS;
```

```
marioP = new Studente("Mario", "Rossi",  
                      "RSSMRA85T10A562S", "La Sapienza, Roma");  
marioS = (Studente)marioP; // OK  
System.out.println(marioS.getUniversita());  
// La Sapienza, Roma
```

```
Persona paoloP;  
paoloP = new Persona("Paolo", "Gialli",  
                     "GLLPLA86R19H148E");  
paoloP = (Studente)paoloP; // NO, ERRORE!!
```

La classe **Object**

- In Java, tutte le classi estendono (direttamente o indirettamente) la classe predefinita **Object**
 - la classe **Object** definisce un comportamento comune per tutti gli oggetti istanza
 - tutte le classi ereditano questo comportamento comune, ma possono ridefinirlo se necessario
- ad esempio, la classe **Object** definisce il metodo *equals()* per verificare se due oggetti sono uguali

```
public boolean equals(Object obj)
```

Il metodo *equals()* di **Object**

- Nell'implementazione di **Object**, due oggetti sono uguali se e solo se sono identici (ovvero, se sono lo stesso oggetto)
- Ogni classe in cui è significativa una nozione di uguaglianza (diversa dall'identità) dovrebbe ridefinire questo metodo

Ereditarietà multipla

- L'**ereditarietà multipla** permette ad una classe di ereditare direttamente da più classi che non siano in qualche rapporto di parentela
 - nella realtà ciò può aver senso; ad esempio, un **Cammello** è sia un **Animale** sia un **MezzoDiTrasporto**
 - alcuni linguaggi di programmazione orientati agli oggetti, come ad esempio il C++, prevedono l'ereditarietà multipla
 - tuttavia risulta difficile da gestire in molti casi
- Java non consente l'ereditarietà multipla
 - una classe può estendere direttamente una sola classe

I modificatori di accesso

- Prima di procedere nello studio dei vari meccanismi di ereditarietà, apriamo una parentesi sui modificatori di accesso
- I **modificatori di accesso** permettono di decidere le regole di accessibilità alle variabili e ai metodi di un oggetto da parte di altri oggetti
 - sono indispensabili per realizzare l’incapsulamento di informazioni “riservate”
 - in Java esistono quattro modificatori di accesso:
 - **private** notazione UML –
 - **public** notazione UML +
 - **protected** notazione UML #
 - **package** notazione UML ~

I modificatori *private* e *public*

- Fino ad ora abbiamo incontrato solo i modificatori *public* e *private*
- **private**: la variabile (o il metodo) a cui si riferisce è accessibile ai soli metodi definiti entro la classe di appartenenza della variabile (metodo)
- **public**: la variabile (o il metodo) a cui si riferisce è accessibile a qualunque metodo (anche se definito in una classe diversa)
- Le variabili di istanza dovrebbero essere dichiarate quanto più possibile *private* (o *protected*) perché definiscono lo stato di un oggetto

I modificatori **private** e **public**

- I metodi definiti come pubblici permettono a chiunque di inviare messaggi all'oggetto, cioè definiscono il comportamento pubblico dell'oggetto:
 - le variabili che rappresentano lo stato dovrebbero essere modificabili o accessibili attraverso metodi pubblici e non direttamente

Il modificatore *protected*

- Spesso è utile permettere a una classe derivata
 - di accedere alle componenti “private” della classe base,
 - senza che queste componenti siano rese accessibili a tutti gli oggetti
- In casi come questo, viene utilizzato il modificatore *protected*. Un componente (variabile e metodo) dichiarato *protected* in una classe **C**
 - può essere acceduto dai metodi definiti in classi che estendono **C**,
 - purché queste classi siano definite nell’ambito dello stesso package in cui viene definito **C** (chiariremo più avanti il concetto di package)

Il modificatore *protected*

- Nella definizione di classi da estendere, viene spesso utilizzato il modificatore *protected* anziché il modificatore *private*
- Il modificatore *package* verrà analizzato più avanti

Classi astratte

- Una **classe astratta** è una classe implementata in modo parziale, in quanto serve a modellare oggetti con un alto livello di astrazione
 - alcuni metodi potrebbero non essere concettualmente implementabili
 - esempio, il metodo *area()* per un oggetto generico di tipo *Forma*
 - tali metodi, detti **metodi astratti**, avranno solo un prototipo, ma non un corpo

Classi astratte

- **Le classi astratte**
 - non possono essere istanziate, perché definite in modo incompleto
 - sono progettate per essere estese da classi che forniscono delle specifiche implementazioni per i metodi astratti
 - sono utili nella definizione di una gerarchia di classi, in cui la super-classe (astratta) è usata per definire
 - le proprietà e
 - il comportamento comune per tutte le classi della gerarchia

Uso di classi astratte – un esempio

- Vogliamo definire delle classi i cui oggetti rappresentano delle forme geometriche
 - ad esempio, le classi **Rettangolo** e **Cerchio** per rappresentare rispettivamente rettangoli e cerchi
- Le proprietà delle forme geometriche sono le seguenti
 - i rettangoli sono caratterizzati da due lati, un nome e un colore
 - i cerchi sono caratterizzati da un raggio, un nome e un colore
 - a un rettangolo si deve poter chiedere il proprio nome, colore, area e perimetro
 - a un cerchio si deve poter chiedere il proprio nome, colore, area e perimetro

La classe astratta *Forma*

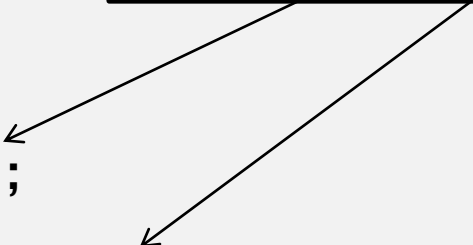
- Definiamo una classe (astratta) *Forma* che definisce le caratteristiche comuni delle classi **Rettangolo** e **Cerchio**
 - il nome
 - il colore
 - i metodi che restituiscono il nome e il colore
 - i metodi che restituiscono l'area e il perimetro

```
public abstract class Forma {  
  
    protected String nome;  
    protected String colore;  
  
    protected Forma(String nome, String colore) {  
        this.nome = nome;  
        this.colore = colore;  
    }  
}
```

... continua dietro ...

La classe astratta *Forma*

```
public String getNome() {  
    return this.nome;  
}  
  
public String getColore() {  
    return this.colore;  
}  
  
public abstract double area();  
  
public abstract double perimetro();  
  
public String toString() {  
    return "Forma di nome " + this.nome + " di colore" +  
        this.colore;  
}  
} // end class Forma
```



il calcolo dell'area e del perimetro
dipende dal tipo specifico di forma
e non può quindi essere definito in
modo generalizzato

La classe Rettangolo

```
public class Rettangolo extends Forma {  
  
    private double lato1;  
    private double lato2;  
  
    public Rettangolo(String nome, String colore,  
                       double lato1, double lato2) {  
        super(nome, colore);  
        this.lato1 = lato1;  
        this.lato2 = lato2;  
    }  
  
    public double area() {  
        return this.lato1 * this.lato2;  
    }  
  
    public double perimetro() {  
        return 2 * (this.lato1 + this.lato2);  
    }  
}
```

... continua dietro ...

La classe Rettangolo

```
public String toString() {  
    StringBuilder strB = new StringBuilder();  
    strB.append("Rettangolo:\n");  
    strB.append("nome: " + this.nome + "\n");  
    strB.append("colore: " + this.colore + "\n");  
    strB.append("lato1: " + this.lato1 + "\n");  
    strB.append("lato2: " + this.lato2 + "\n");  
    strB.append("perimetro: " + this.perimetro() + "\n");  
    strB.append("area: " + this.area() + "\n");  
    return strB.toString();  
}  
} // end class Rettangolo
```

La classe Cerchio

```
public class Cerchio extends Forma {  
  
    private double raggio;  
  
    public Cerchio(String nome, String colore,  
                   double raggio) {  
        super(nome, colore);  
        this.raggio = raggio;  
    }  
  
    public double area() {  
        return Math.PI * Math.pow(this.raggio, 2);  
    }  
  
    public double perimetro() {  
        return 2 * Math.PI * this.raggio;  
    }  
}
```

... continua dietro ...

La classe Cerchio

```
public String toString() {  
    StringBuilder strB = new StringBuilder();  
    strB.append("Cerchio:\n");  
    strB.append("nome: " + this.nome + "\n");  
    strB.append("colore: " + this.colore + "\n");  
    strB.append("raggio: " + this.raggio + "\n");  
    strB.append("perimetro: " + this.perimetro() + "\n");  
    strB.append("area: " + this.area() + "\n");  
    return strB.toString();  
}  
} // end class Cerchio
```

Uso di forme geometriche

```
Rettangolo rect;  
Forma forma;
```

```
rect = new Rettangolo("PippoRect", "Rosso", 10.0, 12.3);  
forma = new Cerchio("PlutoCer", "Verde", 18.42);
```

```
System.out.println(rect.toString());  
// Rettangolo:  
// nome: PippoRect  
// colore: Rosso  
// lato1: 10.0  
// lato2: 12.3  
// perimetro: 22.30  
// area: 123
```


Uso di forme geometriche

```
System.out.println(forma.toString());  
// Cerchio:  
// nome: PlutoCer  
// colore: Verde  
// raggio: 18.42  
// perimetro: 115,73627335824798290496378224002  
// area: 1065,9310776294639225547164344306
```

```
forma = rect;  
System.out.println(forma.toString());  
// Rettangolo:  
// nome: PippoRect  
// colore: Rosso  
// lato1: 10.0  
// lato2: 12.3  
// perimetro: 22.30  
// area: 123
```

Interfacce

- In Java, una **interfaccia (interface)** è una unità di programmazione che consiste nella dichiarazione di un certo numero di metodi d'istanza pubblici che sono implicitamente astratti
 - una interfaccia è simile a una classe astratta che dichiara solo metodi astratti, senza fornire alcuna implementazione
 - come la definizione di una classe, la dichiarazione di una interfaccia definisce un nuovo tipo riferimento, che può essere usato nella dichiarazione di variabili

Implementazione di una interfaccia

- Una classe **implementa (implements)** una interfaccia se implementa (definisce) tutti i metodi dichiarati dall'interfaccia
 - una classe può implementare un numero qualunque di interfacce
 - le interfacce consentono di sopperire alla mancanza di ereditarietà multipla

Esempio – interfaccia *List*

```
public interface List {  
  
    public int size(); /* lunghezza della lista */  
    public boolean isEmpty(); /* verifica se la lista è vuota */  
    public void clear(); /* svuota la lista */  
  
    public void add(Object obj); /* inserisce l'oggetto obj  
        nella testa della lista */  
  
    public boolean contains(Object obj); /* verifica se la lista  
        contiene un elemento uguale a obj */  
  
    public Object remove(Object obj); /* rimuove dalla lista il  
        primo elemento uguale a obj, restituisce l'oggetto rimosso  
        se l'operazione è stata possibile altrimenti null */  
  
    public Object peek(); /* restituisce la testa della lista;  
        pre: la lista è non vuota */  
}
```

... continua dietro ...

Esempio – interfaccia *List*

```
public Object tailPeek(); /* restituisce la coda della  
lista; pre: la lista è non vuota */  
  
public void addToHead(Object obj); /* inserisce l'oggetto  
obj nella testa della lista, esattamente come add() */  
  
public void addToTail(Object obj); /* inserisce l'oggetto  
obj nella coda della lista */  
  
public Object removeFromHead(); /* rimuove e restituisce la  
testa della lista; pre: la lista è non vuota */  
  
public Object removeFromTail(); /* rimuove e restituisce la  
coda della lista; pre: la lista è non vuota */  
  
public Iterator elements(); /* restituisce un  
iteratore per visitare gli elementi della lista */  
  
} // end interface List
```

MyList implements *List*

```
public class MyList implements List {
```

```
    /*  
     * variabili di istanza,  
     * metodi costruttori,  
     * eventuali altri metodi non in List  
     */  
    ...
```

```
    public int size() {  
        ... /* implementazione di size() */  
    }
```

```
    public boolean isEmpty() {  
        ... /* implementazione di isEmpty() */  
    }
```

```
    public void clear() {  
        ... /* implementazione di clear() */  
    }
```

la parola chiave **implements** nella dichiarazione di classe specifica che la classe implementa l'interfaccia *List*

... continua dietro ...

MyList implements *List*

```
public void add(Object obj) {  
    ... /* implementazione di add() */  
}  
  
public boolean contains(Object obj) {  
    ... /* implementazione di contains() */  
}  
  
public Object remove(Object obj) {  
    ... /* implementazione di remove() */  
}  
  
public Object peek() {  
    ... /* implementazione di peek() */  
}  
  
public Object tailPeek() {  
    ... /* implementazione di tailPeek() */  
}
```

... continua dietro ...

MyList implements *List*

```
public void addToHead(Object obj) {  
    ... /* implementazione di addToHead() */  
}  
  
public void addToTail(Object obj) {  
    ... /* implementazione di addToTail() */  
}  
  
public Object removeFromHead() {  
    ... /* implementazione di removeFromHead() */  
}  
  
public Object removeFromTail() {  
    ... /* implementazione di removeFromTail() */  
}  
  
public Iterator elements() {  
    ... /* implementazione di elements() */  
}  
} // end class MyList
```


Osservazioni

- Una volta dichiarata una interfaccia *List*
 - è implicitamente definito un tipo riferimento *List*
- Una variabile riferimento di tipo *List*
 - può referenziare qualunque oggetto
 - a patto che l'oggetto sia un'istanza di una classe che implementa l'interfaccia *List*

Liste di oggetti qualsiasi

- L'interfaccia *List* modella liste di “oggetti qualsiasi”
 - assume che gli elementi della lista siano oggetti di tipo **Object**
- **Vantaggio**
 - una classe che implementa l'interfaccia *List*, ad esempio **MyList**, può essere usata per gestire liste di oggetti qualsiasi
 - liste di oggetti **String**
 - liste di oggetti **Forma**
 - liste di oggetti **Rettangolo**
 - ...

Liste di oggetti qualsiasi

- **Svantaggio**

- una stessa lista, di tipo *List*, può contenere elementi di tipo diverso
 - ad esempio, il primo elemento può essere un tipo **String**, il secondo un tipo **Forma**, il terzo un tipo **JFrame**, ecc.
- l'eterogeneità degli elementi di una lista è quasi sempre indesiderata
 - spesso è causa di errori in fase di esecuzione

Liste di oggetti qualsiasi

- **Domanda:** posso definire un tipo lista contenente elementi di tipo qualsiasi, ma con il vincolo che tutti gli elementi siano dello stesso tipo?
- In altri termini, posso definire un tipo lista i cui elementi siano tutti di uno stesso tipo (non primitivo) generico E?
 - se fosse possibile avrei il vantaggio della genericità, ma eviterei lo svantaggio della eterogeneità

Liste di oggetti qualsiasi

- **Risposta:** Sì, basta utilizzare i **tipi generici (generics)**
 - Java ha introdotto i generics a partire dalla versione 1.5
- Nelle prossime slide si illustreranno alcuni esempi in cui si usano i generics
 - per una descrizione approfondita si rimanda alla documentazione ufficiale Java

Tipi generici – *generics*

- Java consente di parametrizzare (o tipizzare) la definizione di classi e interfacce mediante la specifica di **tipi generici** (**generics**)
 - una classe o interfaccia tipizzata (o generica) modella una famiglia di tipi correlati tra loro
- Ad esempio, un'interfaccia *List* che modelli liste di elementi di uno stesso tipo generico **E** è definibile con la seguente sintassi:

```
public interface List<E> {  
    /* ... Metodi di List ... */  
}
```

Tipi generici – *generics*

- L'interfaccia **List<E>** (da leggersi “lista di **E**”) dichiara una lista generica che può essere utilizzata per un **qualunque tipo** di dato **E** non primitivo
 - **E** è nota come variabile di tipo
 - non esiste una regola che impone l'uso del termine “**E**” per denotare il tipo generico di elemento della lista
 - tuttavia, per convenzione le variabili di tipo sono specificate da nomi costituiti dai seguenti caratteri singoli maiuscoli
 - **E** per un tipo elemento (element)
 - **K** per un tipo chiave (key)
 - **V** per un tipo valore (value)
 - **T** per un tipo generale che non rientra nelle precedenti categorie (type)

Interfaccia *List*<E>

```
public interface List<E> {  
  
    public int size(); /* lunghezza della lista */  
    public boolean isEmpty(); /* verifica se la lista è vuota */  
    public void clear(); /* svuota la lista */  
  
    public void add(E e); /* inserisce l'elemento "e" nella testa della  
        lista */  
  
    public boolean contains(E e); /* verifica se la lista  
        contiene un elemento uguale ad "e" */  
  
    public E remove(E e); /* rimuove dalla lista il  
        primo elemento uguale ad "e", restituisce l'elemento rimosso  
        se l'operazione è stata possibile altrimenti null */  
  
    public E peek(); /* restituisce la testa della lista;  
        pre: la lista è non vuota */  
}
```


Interfaccia *List*<E>

```
public E tailPeek(); /* restituisce la coda della  
lista; pre: la lista è non vuota */  
  
public void addToHead(E e); /* inserisce l'elemento  
"e" nella testa della lista, esattamente come add() */  
  
public void addToTail(E e); /* inserisce l'elemento "e" nella coda  
della lista */  
  
public E removeFromHead(); /* rimuove "e" restituisce la  
testa della lista; pre: la lista è non vuota */  
  
public E removeFromTail(); /* rimuove e restituisce la  
coda della lista; pre: la lista è non vuota */  
  
public Iterator<E> elements(); /* restituisce un  
iteratore per visitare gli elementi della lista */  
  
} // end interface List
```

MyList<E> implements List<E>

```
public class MyList<E> implements List<E> {
```

```
    /*  
        variabili di istanza,  
        metodi costruttori,  
        eventuali altri metodi non in List  
    */  
    ...
```

```
    public int size() {  
        ... /* implementazione di size() */  
    }
```

```
    public boolean isEmpty() {  
        ... /* implementazione di isEmpty() */  
    }
```

```
    public void clear() {  
        ... /* implementazione di clear() */  
    }
```

... continua dietro ...

MyList<E> implements List<E>

```
public void add(E e) {  
    ... /* implementazione di add() */  
}  
  
public boolean contains(E e) {  
    ... /* implementazione di contains() */  
}  
  
public E remove(E e) {  
    ... /* implementazione di remove() */  
}  
  
public E peek() {  
    ... /* implementazione di peek() */  
}  
  
public E tailPeek() {  
    ... /* implementazione di tailPeek() */  
}
```

... continua dietro ...

MyList<E> implements *List<E>*

```
public void addToHead(E e) {  
    ... /* implementazione di addToHead() */  
}  
  
public void addToTail(E e) {  
    ... /* implementazione di addToTail() */  
}  
  
public E removeFromHead() {  
    ... /* implementazione di removeFromHead() */  
}  
  
public E removeFromTail() {  
    ... /* implementazione di removeFromTail() */  
}  
  
public Iterator<E> elements() {  
    ... /* implementazione di elements() */  
}  
} // end class MyList
```

Uso di **MyList<E>** – lista di stringhe

```
MyList<String> lst = new MyList<String>(); /* crea una  
lista di stringhe */
```

```
lst.add("Pippo"); // OK  
lst.add("Pluto"); // OK  
lst.add("Paperino"); // OK
```

```
lst.add(new Integer(12)); /* ERRORE in fase di  
compilazione, il costruttore della lista specifica che  
gli elementi della lista sono stringhe */
```

Esempio – Interfaccia *Comparable*<T>

- Java definisce l'interfaccia *Comparable*<T> che rende possibile l'ordinamento di collezioni di oggetti di un generico tipo T
 - per poter ordinare una collezione di oggetti è necessario che gli oggetti siano “confrontabili”
 - dati due oggetti qualsiasi della collezione deve essere sempre possibile dire “chi precede chi”
 - gli oggetti di classi che implementano l'interfaccia *Comparable*<T> sono confrontabili

Interfaccia *Comparable*<T>

- L'interfaccia *Comparable*<T> dichiara un solo metodo

```
public interface Comparable<T> {  
  
    /* restituisce:  
        - zero, se questo oggetto e obj sono uguali  
        - un valore negativo, se questo oggetto è minore  
          di obj  
        - un valore positivo, se questo oggetto è  
          maggiore di obj */  
    public int compareTo(T obj);  
  
}
```

Interfaccia *Comparable*<T>

- La classe **String** implementa l'interfaccia *Comparable*<**String**>
 - implementa il metodo *compareTo()* considerando l'ordinamento lessicografico delle stringhe
- La classe **Integer** implementa l'interfaccia *Comparable*<**Integer**>
 - implementa il metodo *compareTo()* considerando l'ordinamento naturale degli interi

Esercizio – ordinamento di oggetti

- **Domanda:** Supponiamo di disporre di una collezione di oggetti confrontabili, come posso ordinare la collezione?
- **Risposta:** grazie al polimorfismo posso implementare una volta per tutte un metodo (statico) di ordinamento ed utilizzarlo
 - nel seguito illustreremo come ordinare collezioni di oggetti di tipo *Forma*

Ordinare oggetti di tipo *Forma*

- Per prima cosa è necessario rendere tutti gli oggetti di tipo *Forma* confrontabili
 - è necessario stabilire un criterio di confronto (ordinamento), ad esempio si può utilizzare l'area
 - la classe astratta *Forma* deve implementare l'interfaccia *Comparable*<*Forma*>, cioè
 - deve implementare il metodo *compareTo()* utilizzando l'area quale criterio di confronto

Nuova classe astratta *Forma*

```
public abstract class Forma implements Comparable<Forma> {  
  
    // ... costruttore e altri metodi di Forma ...  
  
    /* Confronta questa forma con un'altra forma "f".  
       Restituisce:  
       - zero, se questo oggetto e "f" hanno la stessa area  
       - un valore negativo, se questo oggetto ha un'area minore  
         (precede) di "f"  
       - un valore positivo, se questo oggetto ha un'area maggiore  
         (segue) di "f". */  
  
    public int compareTo(Forma f) {  
        int risultato = 0;  
        if (this.area() > f.area())  
            risultato = 1;  
        else if (this.area() < f.area())  
            risultato = -1;  
        return risultato;  
    }  
} // end abstract class Forma
```

Ordinare oggetti di tipo *Forma*

- Definiamo una classe **SortArray** per ordinare array (collezioni) di oggetti di tipo *Forma*

```
public class SortArray {  
  
    private static void scambia(Forma[] dati, int i, int j) {  
        Forma temp = dati[i];  
        dati[i] = dati[j];  
        dati[j] = temp;  
    }  
  
    public static void bubbleSort(Forma[] dati) {  
        // implementazione di bubbleSort()  
    }  
  
} // end class SortArray
```

Ordinare oggetti di tipo *Forma*

```
public static void bubbleSort(Forma[] dati) {  
    int n; // numero di elementi da ordinare  
    int i; // indice usato per la scansione di dati  
    int ordinati; // numero di elementi già ordinati  
    int ultimoScambio; // posizione dell'ultimo scambio  
  
    n = dati.length;  
    ordinati = 0;  
    while (ordinati < n-1) {  
        ultimoScambio = 0;  
        for (i = 1; i < n-ordinati; i++)  
            if (dati[i].compareTo(dati[i-1]) < 0) {  
                scambia(dati, i, i - 1);  
                ultimoScambio = i;  
            }  
        ordinati = n - ultimoScambio;  
    }  
} // end method bubbleSort()
```

Classe di test – TestSortArray

```
public class TestSortArray {  
  
    public static void main(String[] args) {  
  
        Forma[] dataArray = new Forma[7];  
  
        dataArray[0] = new Cerchio("PlutoCer", "Verde", 18.42);  
        dataArray[1] = new Cerchio("TizioCer", "Giallo", 5.6);  
        dataArray[2] = new Cerchio("PaperinoCer", "Bianco", 8.54);  
        dataArray[3] = new Cerchio("CaioCer", "Arancione", 21.54);  
  
        dataArray[4] = new Rettangolo("PippoRect", "Rosso", 10.0, 12.3);  
        dataArray[5] = new Rettangolo("MarioRect", "Viola", 7.0, 17.23);  
        dataArray[6] = new Rettangolo("WalterRect", "Marrone", 2.0, 6.54);  
    }  
}
```

... continua dietro ...

Classe di test – TestSortArray

```
System.out.println("Array non ordinato\n");

for (int i = 0; i < dataArray.length; i++)
    System.out.println(dataArray[i].toString() + "\n");

SortArray.bubbleSort(dataArray); // ordina l'array

System.out.println("\nArray ordinato\n");

for (int i = 0; i < dataArray.length; i++)
    System.out.println(dataArray[i].toString() + "\n");
}

} // end class
```

Osservazione

- Il metodo statico *bubbleSort()* della classe **SortArray** consente di ordinare soltanto array di oggetti di tipo *Forma*
 - per ordinare oggetti confrontabili che non sono di tipo *Forma* si deve implementare un nuovo metodo *bubbleSort()*
- **Domanda:** è possibile generalizzare il metodo *bubbleSort()* in modo tale da applicarlo ad una qualsiasi collezioni di oggetti confrontabili?
- **Risposta:** sì, basta utilizzare i metodi generici

SortArray con metodi generici

```
public class SortArray {  
  
    /* metodo generico scambia() */  
    public static <T extends Comparable<T>> void scambia(T[] dati,  
                                                         int i, int j) {  
  
        T temp = dati[i];  
        dati[i] = dati[j];  
        dati[j] = temp;  
    }  
  
    /* metodo generico bubbleSort() */  
    public static <T extends Comparable<T>> void bubbleSort(T[] dati) {  
        // numero di elementi da ordinare  
    }  
  
} // end class SortArray
```

dichiarazione di un tipo generico **T**;
T è un qualsiasi tipo che implementa l'interfaccia
Comparable<T>

SortArray con metodi generici

```
/* metodo generico bubbleSort() */
public static <T extends Comparable<T>> void bubbleSort(T[] dati) {
    int n; // numero di elementi da ordinare
    int i; // indice usato per la scansione di dati
    int ordinati; // numero di elementi già ordinati
    int ultimoScambio; // posizione dell'ultimo scambio

    n = dati.length;
    ordinati = 0;
    while (ordinati < n-1) {
        ultimoScambio = 0;
        for (i = 1; i < n-ordinati; i++)
            if (dati[i].compareTo(dati[i-1]) < 0) {
                scambia(dati, i, i - 1);
                ultimoScambio = i;
            }
        ordinati = n - ultimoScambio;
    }
}
```

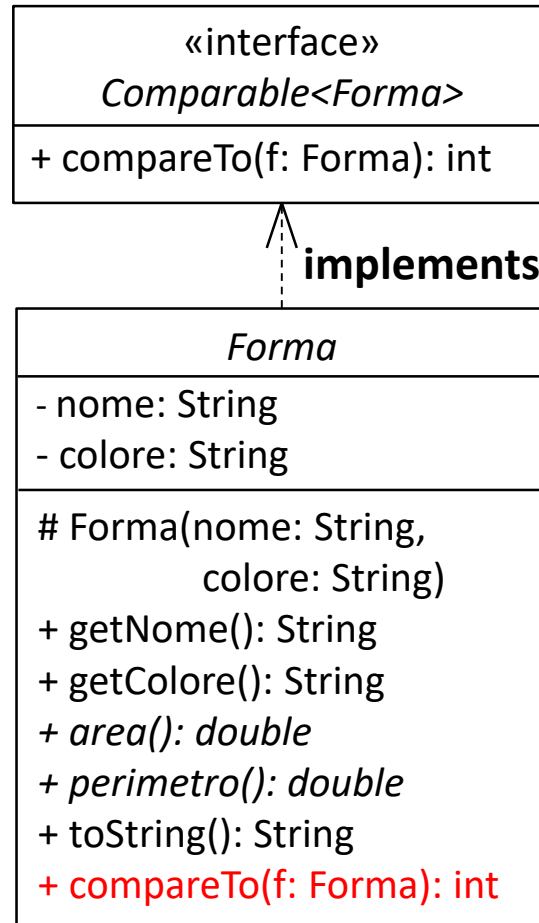
Interfacce e loro implementazione

- Una classe può implementare, direttamente o indirettamente, più di una interfaccia
 - direttamente
 - definendo tutte le funzionalità (metodi) dichiarate dalle diverse interfacce, ed
 - elencando le interfacce implementate dopo la parola *implements*
 - indirettamente
 - estendendo una classe che già implementa una o più interfacce

Interfacce e loro implementazione

- E' anche possibile che
 - una interfaccia sia definita come l'estensione di un'altra interfaccia, e che
 - una classe estenda un'altra classe e contemporaneamente implementi una o più interfacce

Notazione UML di interfacce e loro implementazione



il seguente diagramma
mostra la relazione tra
una interfaccia e una
classe che la implementa

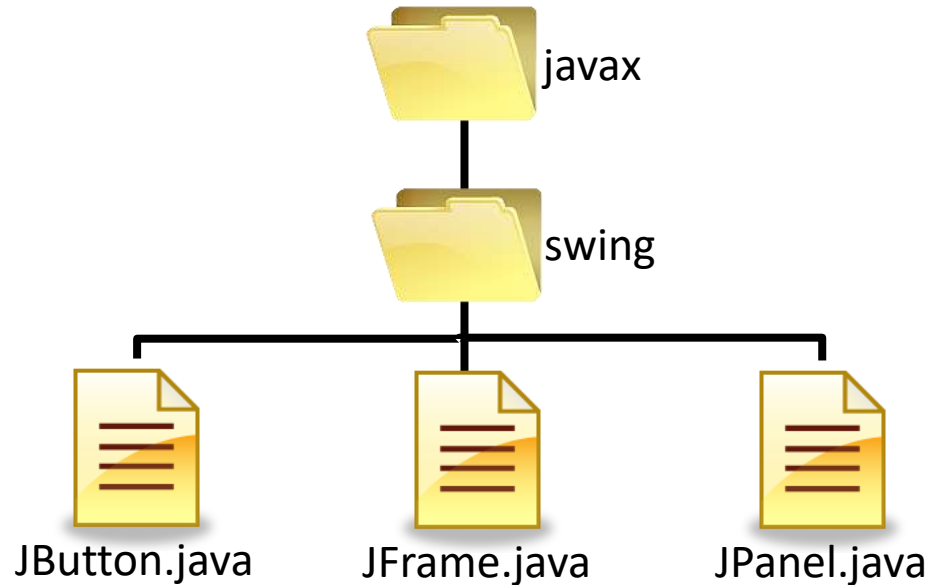
Packages

- Un **package** è un insieme di classi che realizzano funzionalità affini
 - le classi della API di Java sono raggruppate in packages
 - ad esempio il package *java.net* raggruppa funzionalità di rete
 - all'inizio del file in cui si definisce una classe si può specificare il package di appartenenza della classe
 - se non si specifica il package, la classe è assegnata ad un "package di default"

Packages

- I packages possono essere organizzati in strutture gerarchiche
 - ogni *package* può avere dei *sotto-packages* e così ricorsivamente
- Esiste una corrispondenza tra packages e directories:
 - ogni package corrisponde ad una directory con lo stesso nome del package
 - i file delle classi di un package debbono risiedere nella directory associata al package

Packages e directories



`javax.swing.JFrame`

← nome completo che
identifica la classe **JFrame**

`javax.swing.*`

← identifica tutte le classi
pubbliche del package
`javax.swing`

Inclusioni di classi e package

- Se in un file si vogliono utilizzare le funzionalità di una classe o di un intero package già esistenti, occorre importarli
- L'importazione va dichiarata all'inizio del file, prima della dichiarazione della classe, con la seguente sintassi

```
import <nome-package>.<nome-classe>;
```

Inclusioni di classi e package

- Esempi

```
// importa la classe JFrame  
import javax.swing.JFrame;
```

```
// importa tutte le classi del package java.io  
import java.io.*;
```

- Le classi che appartengono allo stesso package si “vedono” senza bisogno di importarsi
- Il package *java.lang* viene sempre importato di default, perché contiene funzionalità di base e di uso comune

Note sull'importazione

- L'importazione (import) di funzionalità in Java è simile al concetto di inclusione (include) del C dal punto di vista logico
- Ci sono tuttavia delle sostanziali differenze:
 - L'include del C comporta l'importazione fisica del codice di tutti i file inclusi nel file che dichiara l'inclusione
 - anche se non tutte le funzionalità incluse sono di fatto utilizzate
 - L'import di Java comporta l'importazione delle sole classi realmente utilizzate tra quelle dichiarate
 - ciò evita inutili crescite del codice (il bytecode è più snello)

Creazione di packages

- Nel file in cui si crea una nuova classe si può definire (in testa al file) il package di appartenenza della classe, la sintassi è:

```
package <nome-package>;
```

- Esempio

```
package lib.geom;

public abstract class Forma {
    // ... codice sorgente di Forma
}
```

Creazione di package

- Se non si dichiara il package di appartenenza della classe, ad essa viene attribuito un package di default (senza nome)
 - in tal caso la classe è visibile solo alle classi della sua stessa directory

La variabile *CLASSPATH*

- **Domanda:** come può sapere il compilatore Java da quale directory inizia la root di uno o più packages?
- **Risposta:** tramite la variabile di ambiente *CLASSPATH*
 - la variabile *CLASSPATH* deve essere pre-impostata a livello di sistema operativo
 - esempio sotto Windows

```
set CLASSPATH = %CLASSPATH%;c:\my_packages\;
```

Ultime osservazioni sui packages

- I packages permettono di organizzare in modo logico le funzionalità di classi eterogenee
- I packages permettono anche di definire delle regole di protezione:
 - solo le classi pubbliche di un package possono essere usate all'esterno del package
 - è possibile che due package differenti definiscano una classe pubblica con lo stesso nome
 - se un file importa due packages che definiscono classi con lo stesso nome, i nomi di tali classi debbono essere disambiguati scrivendo l'intero nome delle classi (cioè il loro path completo)

Il modificatore package

- Il modificatore package dice che l'elemento (classe, variabile o metodo) è visibile solo all'interno del package in cui è definito
 - il modificatore package è il default nel caso in cui non venga specificato un modificatore in modo esplicito per un elemento

Modificatori di accesso

- Riassumiamo le regole di visibilità dei vari modificatori di accesso

modificatore	sottoclasse	package	ovunque
<code>private</code>	NO	NO	NO
<code>protected</code>	SI	NO	NO
<code>public</code>	SI	SI	SI
<code>package</code>	NO	SI	NO

FINE