Unità B2

Gli oggetti: concetti avanzati

Obiettivi

- Comprendere i vantaggi offerti dal meccanismo dell'ereditarietà
- Attivare processi di astrazione e specializzazione

Ereditarietà

- L'ereditarietà permette di **definire nuove classi** partendo da classi sviluppate in precedenza.
- La nuova classe viene definita esprimendo solamente le **differenze** che essa possiede rispetto alla classe di partenza.
- L'ereditarietà permette di specificare "il punto di partenza", cioè la **classe base**, e le **differenze** rispetto a questa.

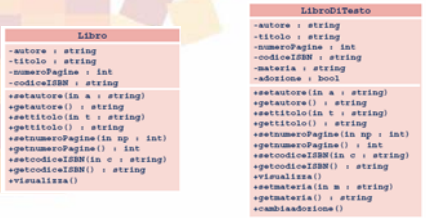
Un esempio

- Classe Animale con proprietà:
  - colore degli occhi
  - peso
  - lunghezza
  - numero dei sensi
  - velocità massima
- Queste proprietà vanno bene per definire uccelli, pesci e mammiferi, però ce ne vorrebbero altre per definire meglio le tre sottocategorie.
- Gli uccelli potrebbero avere la proprietà "apertura alare", ai pesci si potrebbe aggiungere "numero di pinne" e ai mammiferi "lunghezza del pelo".

Sottoclasse e Superclasse

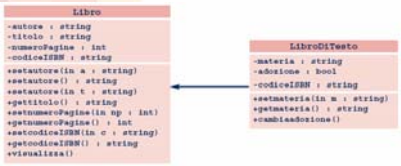
- La nuova classe (nell'esempio Uccello, Pesce e Mammifero) viene definita **sottoclasse** (o **classe derivata**)
- La classe di provenienza (nell'esempio Animale) viene definita **superclasse** (o **classe base**)
- La sottoclasse eredita tutte le caratteristiche (attributi e metodi) della superclasse e si differenza da questa:
  - per l'aggiunta di nuovi attributi e/o metodi
  - per la ridefinizione di alcuni metodi della superclasse
- Attenzione: è vero che un pesce è un animale, non è vero il contrario in quanto un animale non è detto che sia un pesce

Un esempio di ereditarietà



Un diagramma di ereditarietà

- LibroDiTesto **deriva** da Libro e **aggiunge** nuove caratteristiche



Definizione di sottoclassi

- Pseudolinguaggio

```
Classe Sottoclasse Deriva Da Superclasse
  <Attributi >
  <Metodi>
fineClasse Sottoclasse
```

- Java

```
class Sottoclasse extends Superclasse {
  <attributi>
  <metodi>
}
```


La sottoclasse LibroDiTesto

```
class LibroDiTesto extends Libro {

  private String materia;
  private boolean adozione;

  public void setmateria(String materia) {
    this.materia = materia;
  }

  public String getmateria() {
    return materia;
  }

  public void cambiaadozione() {
    adozione = !adozione;
  }
}
```


Istanze di classe e di sottoclasse

- Un oggetto di tipo Sottoclasse è contemporaneamente e automaticamente anche di tipo Superclasse.
- Ogni qual volta che è necessario utilizzare un oggetto di tipo Superclasse è possibile utilizzare un oggetto di tipo Sottoclasse.
- Al contrario invece la regola non vale.
- Ogni oggetto di tipo LibroDiTesto è anche un oggetto di tipo Libro. Infatti è vero che un libro di testo è un libro. Non è vero invece il contrario: un libro non è necessariamente un libro di testo, potrebbe essere per esempio un romanzo.

Polimorfismo

```
Libro lib;
LibroDiTesto libtes = new LibroDiTesto();
lib = libtes;
```

- Il tipo `LibroDiTesto` è compatibile con il tipo `Libro`
- Questo fenomeno è chiamato **polimorfismo** ed è uno dei principi fondamentali della programmazione orientata agli oggetti.
- Una variabile come `lib` definita nell'esempio precedente è polimorfa: può contenere oggetti di tipo diverso.

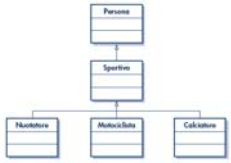
Situazioni d'errore

- Un oggetto della classe base non può essere utilizzato al posto di uno della classe derivata.
- Nel nostro esempio una situazione come la seguente genera un errore:

```
Libro lib = new Libro();
LibroDiTesto libtes;
libtes = lib;     //**** ERRORE ****
```

Gerarchia di classi

- L'ereditarietà può estendersi a più livelli generando quindi una **gerarchia di classi**.
- Una classe derivata può, a sua volta, essere base di nuove sottoclassi.
- Sportivo è sottoclasse di Persona ed è superclasse di Nuotatore, Motociclista e Calciatore.
- Nella parte alta della gerarchia troviamo le **classi generiche**, scendendo aumenta il **livello di specializzazione**.



Un esempio

```
class Persona {
  …
}
class Sportivo extends Persona {
  …
}
class Nuotatore extends Sportivo {
  …
}
class Motociclista extends Sportivo {
  …
}
class Calciatore extends Sportivo {
  …
}
```

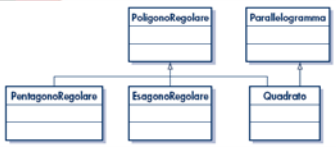

Ereditarietà singola e multipla

- Sono possibili due tipi di ereditarietà:
  - ereditarietà singola
  - ereditarietà multipla
- **L'ereditarietà singola** impone ad una sottoclasse di derivare da **una sola superclasse**.
- L'esempio presentato precedentemente è un caso di ereditarietà singola: ogni sottoclasse ha una sola classe base, mentre è possibile da una superclasse avere più classi derivate.
- Vari linguaggi ad oggetti pongono il vincolo dell'ereditarietà singola per problemi di chiarezza e semplicità d'implementazione, **Java è uno di questi**.
- **Non** è possibile quindi una definizione di classe del tipo:

```
class A extends B,C
```


Ereditarietà multipla

- L'ereditarietà multipla si ha quando una sottoclasse deriva da **più superclassi**
- La classe **`Quadrato`** ha due superclassi: **`PoligonoRegolare`** e **`Parallelogramma`**



Java e l'ereditarietà multipla

- Java **non** prevede ereditarietà multipla fra classi
- L'introduzione delle **Interfacce** (che vedremo in seguito) permette parzialmente di ovviare a questa limitazione
- La definizione di Quadrato in pseudolinguaggio è la seguente:

```
Classe Quadrato Deriva Da PoligonoRegolare,Parallelogramma
    …
fine Classe Quadrato
```

- **`Quadrato`** erediterà attributi e metodi sia da **`PoligonoRegolare`** che da **`Parallelogramma`**

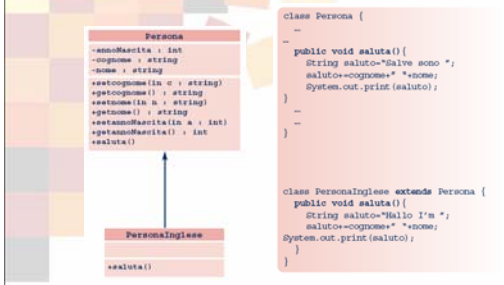
Estensione

- Una classe derivata può differenziarsi dalla classe base aggiungendo nuove caratteristiche:
  - nuovi attributi
  - e/o nuovi metodi
- in questo caso si parla di estensione.
- L'esempio relativo alla classe Libro e LibroDiTesto è un esempio di ereditarietà per estensione: la sottoclasse **aggiunge nuove caratteristiche** ma non altera il comportamento delle funzionalità offerte dalla classe base.

Ridefinizione

- La classe derivata potrebbe però fornire le stesse caratteristiche della classe base differenziandosi invece per il **comportamento**
- Si definisce **ereditarietà per ridefinizione** (**overriding**) la situazione in cui uno o più **metodi** della classe base siano **ridefiniti** nella classe derivata
- I metodi avranno quindi la **stessa firma** (nome e lista di tipi dei parametri) ma **differente corpo**

Un esempio di overriding



Overring e overloading

- Attenzione a non confondere
- il **sovraccarico** dei metodi (**overloading**)
  situazione in cui oltre al corpo del metodo è differente anche la sua firma
- con la **ridefinizione** (**overriding**)
  situazione in cui la firma del metodo è identica ma è differente il corpo

Estensione e ridefinizione

- È possibile incontrare situazioni in cui sono presenti **sia estensione che ridefinizione**
- Nell'esempio presentato in precedenza relativo alla biblioteca scolastica avremmo potuto ridefinire nella classe LibroDiTesto il metodo stampa() per includere la visualizzazione dei nuovi attributi.

super

- La parola chiave **`this`**, intesa come riferimento all'oggetto stesso, è utilizzata per eliminare ambiguità o per esplicitare ulteriormente il riferimento ad un attributo o ad un metodo interno.
- In una gerarchia di classi può essere necessario far riferimento ad un **attributo o metodo della superclasse**, in questo caso si utilizza la parola chiave **`super`**.

Un esempio

```
class Libro {
  …
  public void stampa(){
    System.out.print(autore+" ");
    System.out.print(titolo+" ");
    System.out.print("pag. "+numeroPagine);
    System.out.print(" "+codiceISBN);
  }
  …
}

class LibroDiTesto extends Libro {
  …
  public void stampa(){
    super.stampa();
    System.out.print(" "+materia);
  }
  …
}
```

La classe Object

- Tutte le classi Java sono **implicitamente derivate dalla classe Object** che risulta quindi al vertice di ogni gerarchia di classi.
- La classe Object implementa alcuni metodi che possono ovviamente essere ridefiniti dalle nuove classi.
- In particolare può risultare utile ridefinire il metodo **toString()** che fornisce una rappresentazione della classe sotto forma di stringa.

clone e finalize

- **clone()** serve per poter **clonare un oggetto**, ovvero per crearne un altro uguale
- E' necessario che la classe a cui appartiene l'oggetto implementi l'interfaccia Cloneable (le interfacce saranno trattate in seguito).
- **finalize()** è un metodo che viene chiamato dalla Java Virtual Machine quando viene distrutto un oggetto

final

- Abbiamo fino ad ora utilizzato il modificatore **final** per definire **attributi non modificabili** (costanti)
- final può essere utilizzato anche **per i metodi**; anche in questo caso sta a significare la non modificabilità del metodo *(un metodo dichiarato final non può essere ridefinito da una sottoclasse)*
- Esistono anche **classi** definite **final** che non possono essere estese *(una classe dichiarata final non può avere sottoclassi)*

Un esempio

```
class Parola extends String {
  public boolean isUnSaluto(){
    if (this.equals("ciao"))
       return true;
    else
       return false;
  }
}
```

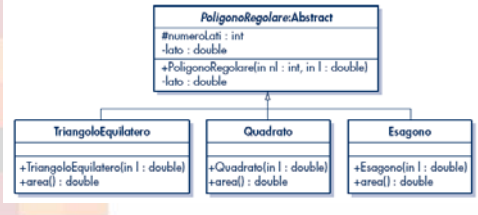
- In questo esempio si tenta di estendere la classe String aggiungendo un nuovo metodo.
- L'esempio non è funzionante in quanto la classe String fornita dal linguaggio è definita final quindi non estendibile.
- L'errore in fase di compilazione è il seguente:

```
Cannot inherit from final java.lang.String
```


Classi e metodi astratti

- Si definiscono **metodi astratti** quei metodi in cui è presente **solo la firma ma non il corpo** (il modificatore di un metodo astratto è abstract)
- Una classe che contiene almeno un metodo astratto si definisce **classe astratta** e da essa **non** possono essere istanziati oggetti
- Le sottoclassi di una classe astratta devono implementare **tutti** i metodi astratti della classe base o essere a loro volta astratte

Un esempio



```
abstract class PoligonoRegolare {
  // Superclasse della gerarchia dei poligoni regolari
  protected int numeroLati; //numero lati poligono
  protected double lato; //lunghezza del lato
  // Costruttore
  PoligonoRegolare(int nl,double l) {
    numeroLati=nl;
    lato=l;
  }

  public double perimetro() {
    return lato*numeroLati;
  }

  abstract double area ();
}

class TriangoloEquilatero extends PoligonoRegolare {

  TriangoloEquilatero(double l) {
    // viene attivato il costruttore della superclasse
    super(3,l);
  }

  // implementazione del metodo astratto della superclasse
  public double area() {
    return (lato*altezza()/2);
  }

  // metodo privato per il calcolo dell'altezza del triangolo
  private double altezza () {
    return (lato*Math.sqrt(3)/2);
  }
}
```

Vantaggi dell'ereditarietà

- L'ereditarietà facilita il **riutilizzo di software** estendendone o ridefinendone caratteristiche e comportamenti; è possibile adattare una classe preesistente alle nuove esigenze.
- Specificare le **differenze** da una classe simile piuttosto che **ridefinire** completamente la classe facilita enormemente lo sviluppo di nuovi progetti **eliminando ridondanza** di codice.
- L'ereditarietà **non** è un meccanismo di **inclusione** del codice di una classe base in una derivata. Non c'è copia di codice, ogni modifica della struttura di una classe base si ripercuote automaticamente nelle sue classi derivate

L'interfaccia verso il mondo esterno

- L'interfaccia di un oggetto è l'insieme delle firme dei suoi metodi
- Se non si vuole dare accesso diretto agli attributi, e se si vuole nascondere l'implementazione di una classe, l'unica cosa che deve conoscere chi utilizza la nostra classe, è l'interfaccia.
- Conoscere l'interfaccia significa sapere quali sono le operazioni, ovvero i metodi, che si possono invocare su un oggetto

Definizione esplicita di interfaccia

- E' possibile definirne un'interfaccia esplicitamente mediante un costrutto specifico

```
Interfaccia Risorsa
  Metodo leggi() Di Tipo Stringa
  Metodo scrivi(buffer Di Tipo Stringa)
  Metodo disponibili() Di Tipo Intero
Fine Interfaccia Risorsa
```

- L'interfaccia di questo esempio definisce le operazioni di un ipotetico oggetto di tipo Risorsa.
- L'interfaccia non essendo una classe implementata non può essere istanziata direttamente.
- Per implementare delle risorse dobbiamo implementare i metodi dichiarati dall'interfaccia Risorsa
- Ma per ogni tipo di risorsa dovranno essere implementati in maniera differente (es. per accedere ad un file si effettueranno operazioni diverse rispetto che per accedere alla rete)

Utilità delle interfacce

- L'interfaccia fornisce uno **schema** di come dovrà essere strutturata la classe: quali metodi dovranno essere presenti
- L'interfaccia **non fornisce l'implementazione** dei metodi ma lascia allo sviluppatore l'onere di scriverli in modo specifico per ognuna delle classi che fa uso dell'interfaccia.
- L'utilizzo di un'interfaccia è utile quando di devono definire i metodi che dovrà possedere un oggetto senza poterne dare un'implementazione univoca per tutte le tipologie di oggetti che faranno uso dell'interfaccia.
- Una classe che vuole fare uso di un'interfaccia si dice che la **implementa**; questo rende obbligatorio **l'implementazione** di **tutti i metodi** definiti nell'interfaccia.

Interfacce in Java

```
public interface Risorsa {
  public String leggi();
  public void Scrivi(String buffer);
  public int disponibili();
}

public class RisorsaFile implements Risorsa {
  public String leggi() {
    // Ritorno il contenuto del file
  }
  public void Scrivi(String buffer) {
    // Scrivo il contenuto di buffer nel file
  }
  public int disponibili() {
    // Ritorno il numero di caratteri disponibili nel file
  }
}
```

Interfacce ed ereditarietà multipla

- Una classe può implementare anche **più di una interfaccia**
- E' possibile implementare un'**interfaccia** e contemporaneamente estendere una **classe**.
- In questo modo si può ovviare al limite di Java di non possedere l'**ereditarietà multipla**.
- Rimane l'inconveniente che utilizzando un'interfaccia per simulare l'ereditarietà multipla è necessario **implementarne tutti i metodi**, rendendo le scelte di progetto difficili e delicate.

Sintesi dell'unità (1)

- Da una **classe base** è possibile derivare più **sottoclassi** che ne **ereditano** metodi e attributi.
- La specializzazione di una sottoclasse può avvenire **aggiungendo** nuovi attributi e metodi o **ridefinendo** alcuni metodi della classe base.
- **L'ereditarietà multipla** consente ad una sottoclasse di avere più superclassi; **l'ereditarietà singola** pone il limite di una sola superclasse.
- **Java** permette soltanto l'ereditarietà **singola**, anche se con il meccanismo delle **interfacce** fornisce una possibile implementazione dell'ereditarietà **multipla**.

Sintesi dell'unità (2)

- L'ereditarietà può essere utilizzata a più livelli definendo così **gerarchie di classi** che partono da una classe base per estendersi a classi sempre più specifiche.
- In Java ogni classe deriva implicitamente dalla classe **Object**.
- In una gerarchia di classi possono essere presenti **classi astratte** in cui alcuni **metodi** sono **astratti**.
- Il modificatore **final** vieta la ridefinizione di un metodo o la derivazione di una nuova classe.
- Le **interfacce** consentono di definire uno **schema** con cui dovranno essere implementate le classi.
- Una classe che implementa un'interfaccia deve **implementare tutti i metodi** in essa definiti.
- L'**interfaccia** è un metodo da poter utilizzare al posto dell'**ereditarietà multipla**.