Gerarchia di classi Java

Programmazione
Corso di laurea in Informatica

Specializzare un modello

- Una classe modella un pezzo di realtà
  - Una macchina, una moneta, un impiegato
- Spesso è necessario specializzare la versione originale del modello
  - Un dirigente è un particolare impiegato
  - **Dichiarazione di classi che illustra l'idea**
- Occorre evitare la duplicazione dei dati
- Alcuni metodi sono utili per entrambe le classi
- **Specializzare le classi e l'ereditarietà**

Ereditarietà

- Consiste nel definire una classe tramite un'altra già definita
- In Java, per stabilire una relazione di sottoclasse, si usa la parola riservata `extends`

```
public class Dirigente extends Impiegato
{
    public String divisione;
}
```


Ereditarietà

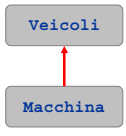
- Il poter definire una struttura gerarchica tra le classi costituisce una fondamentale tecnica dell'approccio object-oriented
- facilita il disegno del software e ne promuove il riuso
  - Derivare nuove classi da già definite
  - Creare una gerarchia tra classi
  - Controllare l'ereditarietà con il modificatore `protected`
  - Polimorfismo tramite ereditarietà

Ereditarietà

- La classe esistente viene detta ***classe antenato*** o ***superclasse*** o anche ***classe base***
- La nuova classe derivata viene detta ***classe figlio*** o ***sottoclasse***.
  - Come i nomi suggeriscono, il figlio eredita alcune caratteristiche dal genitore o antenato
  - La ***sottoclasse*** eredita metodi e dati dalla ***superclasse***

Ereditarietà

- Nel *diagramma di classe* si mette in evidenza la relazione tra classe e sottoclasse introdotta dall'ereditarietà
  - la freccia è rivolta verso la superclasse
  - L'ereditarietà implementa una relazione ***is-a***, cioè il figlio **è** una versione più specifica del padre



Definire sottoclassi

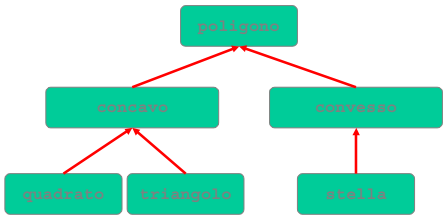
- L'ereditarietà è appropriata ogni volta che esiste una relazione **è_*un*** o **is_a** tra due oggetti.
  - Il dirigente è un particolare impiegato
  - Ha senso definire la classe `Dirigente` come una ***sottoclasse*** di `Impiegato`
  - La classe `Dirigente` eredita dalla classe `Impiegato` alcune caratteristiche e funzionalità

Definire sottoclassi

- Un quadrato è un particolare rettangolo, che è un tipo di poligono
  - La classe `Quadrato` estende la classe `Rettangolo`, sottoclasse di `Poligono`.
- Un taxi è una particolare macchina che è un particolare veicolo
  - `class Taxi extends Macchina { … }`
  - A sua volta la classe `Macchina` è_un particolare `Veicolo`
    `class Macchina extends Veicolo { … }`
- Un `Dizionario` è un tipo di `Libro`
  - `Libro.java`, `Dizionario.java` con `TestDizionario.java`

Gerarchie di classi

- Una sottoclasse può essere superclasse di un proprio discendente
  - Così si forma una ***gerarchia di classi***



Gerarchie di classi

- Due discendenti della stessa superclasse si dicono ***fratelli***
- Un buon disegno di relazioni tra classi mette tutte le caratteristiche comuni il più alto possibile in gerarchia
- Un membro ereditato viene passato verso il basso nella linea di discendenza
- La gerarchia spesso deve essere estesa e modificata per soddisfare esigenze che cambiano
- Non esiste **la** gerarchia appropriata a tutte le situazioni

Ereditare dalla superclasse

- La sottoclasse eredita dalla superclasse e da tutti gli antenati. In particolare eredita:
  - Tutti i membri (variabili e metodi) della superclasse **accessibili** a quella sottoclasse, a meno che
  - Esplicitamente nasconda le variabili o ne sovrascriva i metodi, ridefinendoli.
- I costruttori non sono membri di una classe e perciò non vengono ereditati dalle sottoclassi.

I membri accessibili

- Una sottoclase eredita i membri **accessibili** della superclasse, dichiarati:
  - `public` o `protected` nella superclasse
  - di default, senza modificatore, purchè siano nello stesso pacchetto della superclasse
- Una sottoclasse **non** eredita i membri dichiarati
  - `private` dalla superclasse
  - con lo stesso nome nella sottoclasse
    - Variabili oscurate
    - Metodi sovrascritti

Variabili oscurate

```
class Sopra {
      Number unNumero;
}
class Sotto extends Sopra {
      Float unNumero;
}
```

- La variabile `unNumero` nella classe `Sotto` oscura la visibilità della variabile `unNumero` della classe `Sopra`

I modificatori di visibilità

- Servono per controllare l'ereditarietà
  - Determinano quali membri della classe vengono o non vengono ereditati
- Si ereditano le variabili e i metodi dichiarati con visibilità `public`, ma non quelli con visibilità `private`
  - Le variabili `public` violano la regola di incapsulare i dati
  - Quando si vuole stabilire una relazione di ereditarietà possiamo usare un terzo modificatore di visibilità: `protected`

Il modificatore `protected`

- Il modificatore di visibilità `protected` consente ai figli di ereditare i membri di una classe base
- Il modificatore `protected` favorisce l'incapsulamento più del modificatore `public`
- Ma il modificatore `protected` non contribuisce all'incapsulamento in modo stretto come `private`
- I dettagli in appendice F del testo Lewis-Loftus

Il riferimento super

- Il riferimento `super` viene usato per riferirsi alla superclasse.
  - Per accedere a variabili, a metodi o a costruttori della superclasse
  - Spesso occorre usare il costruttore dell'antenato per inizializzare i campi di un oggetto figlio, relativi all'antenato, cioè ereditati dalla superclasse

Il riferimento super

- Tramite il riferimento `super` si possono accedere variabili oscurate dalla sottoclasse
  - La variabile `unNumero` della classe `Sopra` è accessibile anche dalla classe `Sotto`

    `super.unNumero`
- O ai metodi sovrascritti della superclasse

Sovrascivere metodi

- Una classe figlio può *sovrascrivere* la definizione di un metodo ereditato per specializzarlo
  - Un figlio può dover modificare un metodo ereditato
- Il nuovo metodo deve avere **lo stesso prototipo** del metodo della superclasse, ma un diverso corpo
- Il tipo dell'oggetto a cui è inviato il metodo determina la versione del metodo invocato

Sovrascivere metodi

- Il metodo della superclasse può essere esplicitamente invocato mediante il riferimento `super`
- Se un metodo è dichiarato anche con il modificatore `final` allora non può essere sovrascritto
- La sovrascrittura può essere applicata a variabili
  - si dice che le variabili sono ***oscurate***
- `Disinfettante.java`, e la sottoclasse `Detergente.java`
- `Massima.java`, `Consiglio.java` con `Messaggi.java`

Regole per sovrascrivere metodi

- Il prototipo dei due metodi deve essere identico: stesso tipo di rientro e parametri
- Il metodo sovrascritto non può essere meno accessibile del metodo originale
- Un metodo sovrascritto non può sollevare differenti tipi di eccezioni rispetto al metodo originale

Costruttori della sottoclasse

- I costruttori non vengono ereditati, non essendo membri di classe
- Vengono definiti esplicitamente dal programmatore
- Oppure viene usato quello di default
  - Il costruttore di default è quello senza argomenti fornito automaticamente quando non ne avete definito uno

Chiamata al costruttore della sottoclasse

- Quando un oggetto viene instanziato, viene generata una sequenza d'azioni:
  1. Viene allocato lo spazio per il nuovo oggetto sullo heap. Tutte la variabili d'istanza, anche quelle ereditate vengono inizializzate ai valori di default.
  2. Si esegue la chiamata esplicita a un costruttore della stessa classe (`this()`) se c'è
  3. Altrimenti si invoca il costruttore esplicito o implicito della superclasse e si inizializzano le variabili d'istanza nell'ordine in cui appaiono nel codice.
  4. Si esegue il resto del codice del costruttore
  5. I passi da 2. a 4. vengono ripetuti ricorsivamente per tutte le classi della gerarchia.

Ordine della chiamata al costruttore

- Java richiede che sia completamente inizializzato l'oggetto che descrive il padre prima che si possa eseguire una qualunque computazione sul figlio
- Chiamata implicita al costruttore della superclasse
  - `Arte.java`, `Disegno.java` e `Schizzo.java`
- Chiamata esplicita al costruttore della superclasse tramite `super`
  - `Libro_2.java`, `Dizionario_2.java` con `TestDizionario_2.java`

Costruzione dell'oggetto della sottoclasse

- Di default ogni costruttore di sottoclasse chiama il costruttore senza argomenti del padre
- Se il padre non ne ha uno la sottoclasse non viene compilata
- Se la superclasse ha costruttori con parametri, si invoca `super` con parametri appropriati per controllare esplicitamente la costruzione dell'oggetto padre
- La chiamata a `super` deve essere la prima istruzione del costruttore

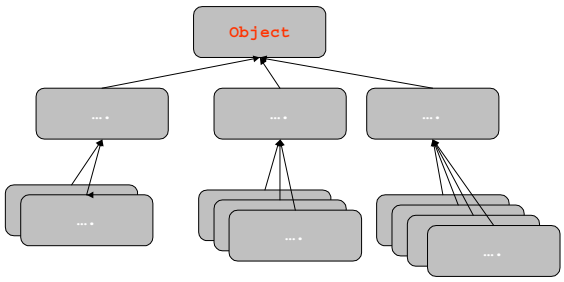
Ereditarietà singola e multipla

- Java fornisce solo ***ereditarietà singola***
  - La sottoclasse deriva da una sola superclasse
- ***L'ereditarietà multipla*** consente la derivazione di una classe da più classi antenati, e di ereditare i membri di più classi
- L'ereditarietà multipla introduce ambiguità, che devono essere risolte
  - Ad esempio: lo stesso nome di variabile in più classi
- Nella maggior parte dei casi, l'uso di *interfacce* supera i limiti dell'ereditarietà singola senza le complicazioni di quella multipla

Sovraccaricare vs. sovrascrivere

- I concetti di ***sovraccaricamento*** e di ***sovrascrittura*** sono diversi
- ***Sovraccaricare*** si riferisce alla possibilità di definire più metodi nella stessa classe con lo stesso nome ma **segnature diverse**
  - Il sovraccaricamento consente di definire operazioni simili in modi diversi con dati diversi
- ***Sovrascrivere*** si riferisce a due metodi, uno della superclasse e uno della sottoclasse, che devono avere lo **stesso prototipo**
  - La sovrascrittura consente di definire operazioni simili in modi diversi per oggetti diversi

Diagramma delle classi



La classe Object

- La classe `Object` è predefinita nel pacchetto `java.lang` della libreria standard di classi
- Tutte le classi discendono dalla classe `Object`
- Se una classe non è esplicitamente dichiarata discendente da una superclasse, si assume che discenda dalla classe `Object`
- La classe `Object` è la radice di tutti gli alberi di discendenza

La classe Object

- La classe `Object` contiene alcuni metodi, pochi ma utili, che vengono ereditati da tutte le classi
- Il metodo `toString` ad esempio è definito per la classe `Object`
  - Il metodo `toString` della classe `Object` restituisce una stringa di caratteri contenete il nome della classe e un valore che identifica l'oggetto univocamente
- Ogni volta che definiamo esplicitamente `toString`, in effetti lo riscriviamo

La classe Object

- Il metodo `println` richiama il metodo `toString` per ogni oggetto che gli viene passato come parametro
  - perché tutti gli oggetti hanno un metodo `toString` ottenuto per ereditarietà
- Il metodo `equals` della classe Object determina se due riferimenti sono *alias*
  - Spesso è necessario riscrivere `equals` per definire il metodo con una semantica appropriata
- `Studente.java`, `StudenteDott.java` con `Accademia.java`

Decomposizione di metodi e ereditarietà

- Nella decomposizione di metodi spesso si usano metodi di servizio richiamati all'interno di un metodo, in genere dichiarati `private`
- Un problema se il metodo di servizio deve essere ereditato e sovrascritto
- `Figure.java` superclasse `Triangle.java`, `Box.java` e `Diamond.java` `Driver.java`

Classi astratte

- Una classe incompleta che contiene metodi astratti
- È un segnaposto nella gerarchia delle classi, modella un concetto generico
  - Il modificatore `abstract` nell'intestazione della classe
- Una classe `abstract` diventa incompleta per definizione
- La classe `abstract` non può essere instanziata ma deve essere specializzata

Classi astratte

- Una classe deve essere dichiarata astratta se:
  - Contiene la dichiarazione di un metodo astratto
  - Eredita un metodo astratto dalla superclasse e non ne fornisce l'implementazione
  - Dichiara di implementare un interfaccia, ma non fornisce l'implementazione di tutti i metodi

Classi astratte

- Il discendente di una classe astratta deve sovrascrivere i metodi astratti della superclasse da cui discende o sarà considerata anche lei astratta
- Un metodo astratto non può essere definito `final`, `static` o `private`
  - Perché dovrà essere sovrascritto (final)
  - Perchè non ha ancora una definizione (static)
  - Perchè non potrebbe essere ereditato (private)
- Una questione di scelta di disegn
  - Aiuta a stabilire gli elementi comuni che dovranno avere le sottoclassi di una classe che rappresenta un concetto astratto

Classe astratte – sintesi

- Una classe astratta è definita con il modificatore `abstract`
- Può contenere metodi implementati e costruttori.
- Se ha almeno un metodo astratto allora deve essere dichiarata astratta.
- Non è possibile istanziarla. Spesso dichiara una variabile del tipo classe astratta, in cui poi si archivia il riferimento a oggetti delle sottoclassi
- Le sottoclassi devono fornire implementazione per tutti i metodi astratti o anche loro saranno considerate astratte.
- I metodi astratti non possono essere dichiarati con i modificatori `private`, `static` o **final**.

Quando sono appropriate

- Non si devono istanziare oggetti della classe. La dichiarazione `abstract` garantisce che non verrà mai istanziata.
- I dati comuni possono essere raccolti in una superclasse, anche senza metodi.
  - Qui si usa una gerarchia di classi in cui una classe serve per scambiare dati ma non ci sono metodi comuni
- Iniziare una gerarchia di classi che devono essere specializzate ulteriormente
  - Qui si usa la classe astratta per contenere metodi la cui specializzazione è rinviata alle classi specializzate
  - Definire il comportamento fondamentale della classe senza darne l'implementazione

Riferimenti ed ereditarietà

- Tramite riferimenti si può accedere a un oggetto legato alla classe da una relazione di ereditarietà
- Se la classe `Festa` ha una sottoclasse `Natale`, allora una variabile a riferimento della classe `Festa` può anche essere usata per puntare a un oggetto `Natale`


```
Festa giorno;
giorno = new Natale();
```


Riferimenti e ereditarietà

- Nella gerarchia di classi la sottoclasse specializza la superclasse e introduce nuove funzionalità
- Ma la nuova classe può essere vista come un tipo di quella esistente
- Questo consente l'upcasting
  - La nuova classe è costituita dagli elemneti della classe base cui aggiunge i propri
  - L'upcasting è sempre lecito


```
class Strumento {
      public void suona() {…}
      static void brano (Strumento s) {
            …
            s.suona()
      }
}
Class Archi extends Strumento {
      …
      public static void main(String[] a) {
            Archi violino = new Archi();
            Strumento.brano(violino);
      }
}
```

Al metodo di classe `brano` viene passato un parametro di tipo `Archi` e non di tipo `Strumento`.
Il riferimento a `Archi`, `violino`, viene convertito a un riferimento a `Strumento`, mediante un ***upcasting***.

Riferimenti ed ereditarietà

- Possiamo assegnare a variabili a riferimento di una superclasse un oggetto di una sottoclasse, semplicemente con un assegnamento
  - Conversione larga o upcasting
- Assegnare a un oggetto discendente il riferimento a un antenato può essere fatto, ma richiede un ***cast*** esplicito
  - Conversione stretta o downcasting
- La conversione larga è più utile e meno pericolosa

Polimorfismo via ereditarietà

- Un *riferimento polimorfico* punta a oggetti di tipo diverso in tempi diversi
- Le interfacce possono essere usate per creare riferimenti polimorfici
- L'ereditarietà può essere usata come base del polimorfismo
  - Una variabile a riferimento può puntare a un oggetto in un dato istante e puntarne a un altro (legato al primo nella struttura gerarchica delle classi) in un altro momento

Polimorfismo via ereditarietà

- Sia `celebrare` un metodo della classe `Festa`, sovrascritto dalla classe `Natale`
- L'invocazione

```
giorno.celebrare();
```

produce effetti diversi:

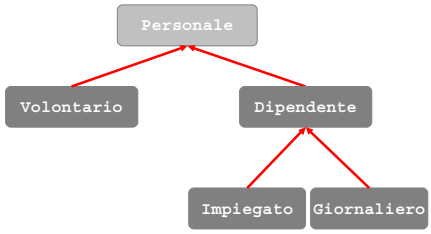
  - Se `giorno` si riferisce a un oggetto della classe `Festa`, viene invocata la versione `celebrare()` della classe `Festa`; se invece punta a un oggetto della sottoclasse `Natale` verrà invocata la versione specifica e sovrascritta della sottoclasse

Polimorfismo via ereditarietà

- È il tipo dell'oggetto cui si punta, che determina quale metodo viene invocato
- Si noti che, se l'invocazione avviene in un ciclo, la stessa riga di codice potrebbe eseguire metodi diversi a ogni passo
- I riferimenti polimorfici vengono risolti a ***run-time***, e non durante la compilazione

Polimorfismo via ereditarietà

- Data la seguente struttura gerarchica:



Polimorfismo via ereditarietà

- Si risolve il compito di calcolare la retribuzione …

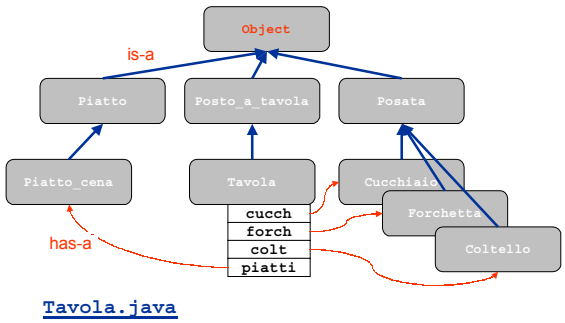
- `Istituzione.java` driver con il main
- `Staff.java` inizializza la lista del personale
- `Personale.java` la classe astratta
- `Dipendente.java` modella un dipendente generico
- `Volontario.java` modella un volontario
- `Impiegato.java` modella un impiegato che può prendere gratifiche, specializzando un dipendente
- `Giornaliero.java` modella un lavoratore a giornata, specializzando un dipendente generico

Accesso indiretto

- Tutti i membri della superclasse sono accessibili alla sottoclasse, anche se non sono ereditati
- I membri di classe ereditati sono accessibili direttamente
- I membri non ereditati sono accessibili indirettamente tramite i metodi ereditati della superclasse

- `Calorie.java` driver, istanzia una Pizza
- `Cibo.java` istanzia un oggetto con dati grammi di grasso e calcola le calorie dovute al grasso
- `Pizza.java`

Composizione di classi vs ereditarietà



`Tavola.java`

Gerarchie di interfacce

- Il meccanismo dell'ereditarietà può essere applicato alle interfacce
  - Un'interfaccia può diventare l'antenata di un'altra
  - L'interfaccia figlio eredita tutti i metodi astratti dal genitore
- Una classe che implementa un'interfaccia derivata deve definire tutti i metodi dell'intarfaccia genitore oltre a quelli del figlio
- Le gerarchie di classi e quelle di interfacce sono distinte