Concetti di Programmazione in Java

Antonio Caliò

Cooperativa Servizi & Formazione
Catanzaro (CZ)

Outline

Presentation agenda

Value vs Reference

Ereditarietà, dynamic binding, polimorfismo

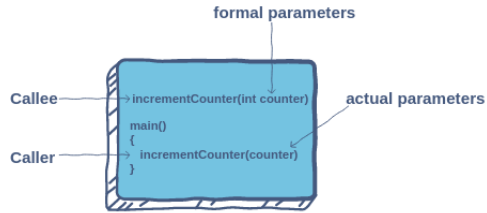
Classi Astratte e Interfaccia

Tipi di dati astratti

Collection framework

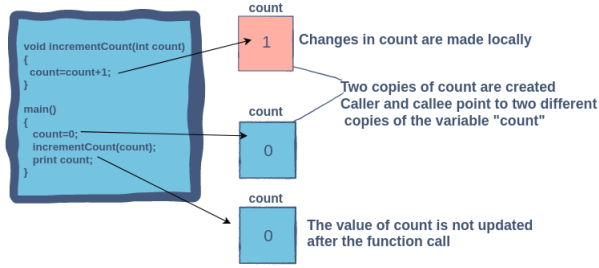
Concetti Chiave

- Chiamata vs Chiamante
  - La funzione da cui parte la chiamata è detta: Chiamante (Caller)
  - L'altra funzione richiamata è detta: Chiamata (Callee)
- Actual vs Formal Parameters
  - Actual Parameters: valori concretamente passati in input durante la chiamata
  - Formal Parameters: valori richiesti nella definizione della funzione



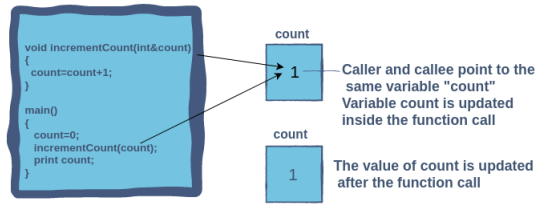
Passaggio per Valore

- Si esegue una copia dei parametri passati in input
  - La funziona chiamante e quella chiamata hanno due set di variabili indipendenti aventi lo stesso valore
  - Le modifiche a tali variabili eseguite dalla funzione chiamata non sono visibili dalla funzione chiamante



Passaggio per Riferimento (o per Indirizzo)

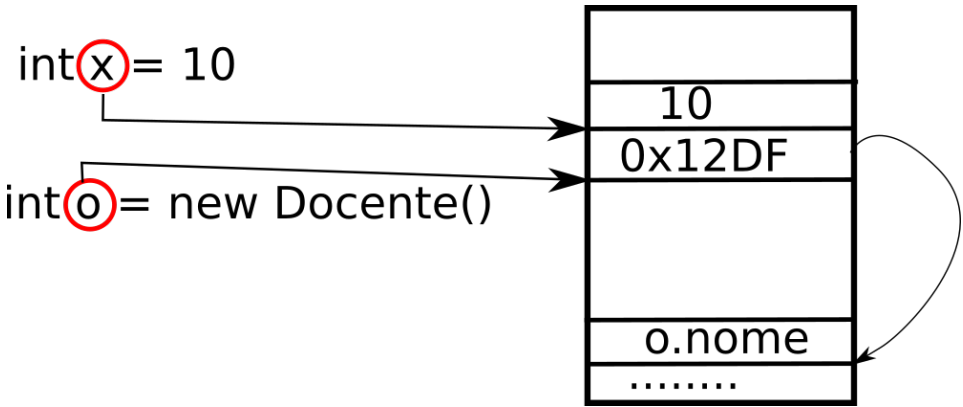
- Il chiamante passa il riferimento i.e., indirizzo di memoria
  - Se all'interno della funzione chiamata si eseguono delle modifiche agli *actual parameters* passati in input:
    - Le modifiche saranno visibili anche dall'esterno della funzione chiamata



Cosa succede in Java?

- In Java i parametri sono sempre passati per valore!
- Tuttavia dobbiamo fare attenzione quando lavoriamo con gli oggetti:
  - Se un metodo richiede in input un oggetto (quindi un tipo non primitivo):
    - Java eseguirà una copia del riferimento a quel determinato oggetto
  - Concretamente, gli oggetti sono passati per riferimento

Cosa succede in Java?



Quiz

```
public class App {
public static void main(String... doYourBest) {
    Simpson simpson = new Simpson();
    transformIntoHomer(simpson);
    System.out.println(simpson.name);
}
static void transformIntoHomer(Simpson simpson) {
    simpson.name = "Homer";
}
}
class Simpson {
String name;
}
```

Quiz

```
public class PrimitiveByValueExample {

public static void main(String... primitiveByValue) {
    int homerAge = 30;
    changeHomerAge(homerAge);
    System.out.println(homerAge);
}

static void changeHomerAge(int homerAge) {
    homerAge = 35;
}
}
```

Oggetti Immutabili

- Oggetti contrassegnati come final
- Una volta inizializzati, il loro valore non può essere modificato
  - Mantengono lo stesso valore per tutta l'esecuzione del programma
- Java ha molte classi immutabili:
  - Integer, Double, Float, Long, Boolean, BigDecimal, String

```
public class StringValueChange {
public static void main(String[] args) {
    String name = "";
    changeToHomer(name);
    System.out.println(name);
}

static void changeToHomer(String name) {
    name = "Homer";
}
}
```

Test

```
public class DragonWarriorReferenceChallenger {
  public static void main(String... doYourBest) {
    StringBuilder wProf =
new StringBuilder("Dragon ");
    String wWeap = "Sword ";
    changeWarriorClass(wProf, wWeap);
    System.out.println("Warrior=" +wProf +
       " Weapon=" + wWeap);
  }
  static void changeWarriorClass(StringBuilder prof,
 String weap) {
    prof.append("Knight");
    weap = "Dragon " + weap;

    weap = null;
    prof = null;
  }
}
```

1. Warrior=null Weapon=null
2. Warrior=Dragon Weapon=Dragon
3. Warrior=Dragon Knight Weapon=Dragon Sword
4. Warrior=Dragon Knight Weapon=Sword

Presentation agenda

Value vs Reference

Ereditarietà, dynamic binding, polimorfismo

Classi Astratte e Interfaccia

Tipi di dati astratti

Collection framework

Nozioni Preliminari

- Progettare una nuova classe per estensione di una classe esistente, dunque per differenza.
  - permette di concentrarsi sulle novità introdotte dalla nuova classe
  - favorisce produttività del programmatore

Una Classe ContoBancario: Specifiche

- Di seguito si considera una classe ContoBancario che definisce le usuali operazioni di deposito e prelievo
- Un conto è identificato da un numero espresso mediante una String, e si caratterizza per il suo bilancio
- Non è permesso al bilancio di andare "in rosso"
  - ossia un prelevamento oltre il valore del bilancio non viene consentito
    - A questo scopo il metodo preleva() ritorna un valore boolean che è true se l'operazione si conclude con successo, false altrimenti
- Metodi accessori permettono di conoscere il numero di conto e il valore corrente del bilancio.

Una classe ContoBancario: Implementazione

```
import java.io.*;
public class ContoBancario{
  private String numero;
  private double bilancio=0;
  public ContoBancario( String numero ){...}//primo costruttore
  public ContoBancario( String numero, double bilancio){..{}//secondo costruttore

public void deposita( double quanto ){ ..}

public boolean preleva( double quanto ) { .. }

public double saldo(){ return bilancio;}
public String conto(){ return numero; }
public String toString(){
   return String.format( "conto=%s bilancio=E %1.2f", numero, bilancio );
  }//toString
}//ContoBancario
```

Un Secondo Conto Bancario, con Fido: Specifiche

- ContoBancario va bene per i clienti “ordinari"
- La banca dispone di un altro tipo di conto ContoConFido riservato a clientela selezionata
  - ammette l'andata in rosso controllata da un fido.
- ContoConFido mantiene molte caratteristiche di ContoBancario ma in più introduce delle differenze:
  - Il bilancio può andare in rosso

Un Secondo Conto Bancario, con Fido: Implementazione

```
import java.io.*;
public class ContoConFido extends ContoBancario {
  private double fido=1000; //default
  public ContoConFido( String numero ) { super( numero );}
  public ContoConFido( String numero, double bilancio ){super( numero, bilancio ); }
  public ContoConFido( String numero, double bilancio, doublé fido ){
    super( numero, bilancio );
    this.fido=fido;
  }
  public boolean preleva( doublé quanto ){ super.preleva(quanto)... }
  public double fido(){ ...}
public void nuovoFido( double fido ){...}
public String toString(){ ... }

}//ContoConFido
```

Il Pronome Super

- Serve a riferirsi alla super classe
  - ad esempio per invocare esplicitamente un costruttore della super classe
    - si delega parte del processo di costruzione.
    - se è usato per questi scopi, super, deve essere la prima istruzione del costruttore.
- Si noti che:
  - essendo private il campo bilancio di ContoBancario: ogni sua modifica va ottenuta mediante i metodi di ContoBancario

Modifiche alla classe ContoConFido

- Modificare la classe ContoConFido di modo che:
  - Si tenga traccia dell'ammontare scoperto da parte del correntista

Esempio

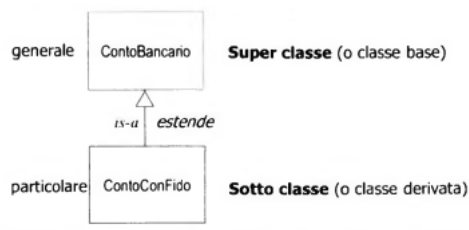
- Il correntista possiede 100$ e cerca di prelevare 200$:
  - L'ammontare scoperto è pari a 100$

Relazione di ereditarietà

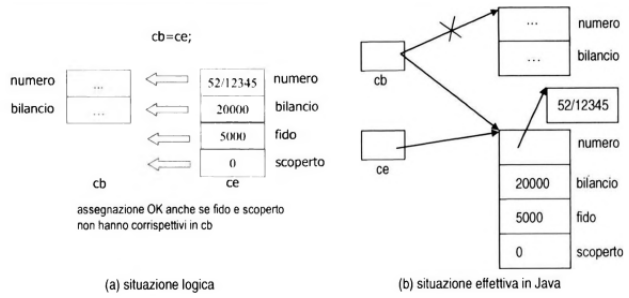
- ContoConFido è-un (is-a) ContoBancario, ma un pò più specializzato.
- ContoConFido è una sottoclasse (o classe derivata)
- ContoBancario una super-classe (o classe base).
- La relazione di ereditarietà da ContoConFido a ContoBancario è una relazione di generalizzazione
- La relazione di ereditarietà è ben definita se un oggetto della classe derivata può sempre sostituire un oggetto della classe base
  - principio di sostituibilità dei tipi
- Tuttavia: un conto bancario non è un conto con fido!!!!

- La parentela ci permette di scrivere:

```
ContoBancario cb=new ContoBancario(...);
ContoConFido ce=new ContoConFido(...);
cb=ce; //assegnazione dal particolare al generale OK
```



Assegnazione tra oggetti come proiezione



- L'assegnazione da particolare a generale corrisponde, ad es., alla proiezione di un punto dello spazio cartesiano (con coordinate x, y e z) sul piano X-Y (la coordinata z è ignorata).
- Nella situazione effettiva di Java, a seguito dell'assegnazione cb=ce, cb punta all'oggetto composito riferito da ce
- Tuttavia, cb lo vede con gli "occhiali" imposti dalla sua classe di appartenenza ContoBancario.
- Pertanto i campi fido e scoperto, anche se effettivamente presenti nell'oggetto puntato da cb, sono ignorati.

Tipo statico e dinamico di un oggetto

```
ContoBancario cb=new ContoBancario(...);
ContoConFido ce=new ContoConFido(...);
cb=ce; //assegnazione dal particolare al generale OK
```

- Dopo l'assegnazione `cb=ce`, ogni uso di `preleva()` si riferisce alla sotto classe
  - `cb` ha tipo statico (legato cioè alla dichiarazione) `ContoBancario`
  - `cb` ha tipo dinamico (guadagnato in seguito all'assegnazione) `ContoConFido`
- Il tipo statico dice cosa si può fare su `cb`
- Il tipo dinamico dice quale particolare metodo va in esecuzione:
  - se uno della super classe o uno della sotto classe.
- Prima dell'assegnazione, `cb.preleva(...)` si riferisce al metodo della super classe.
- Dopo l'assegnazione, `cb.preleva(...)` invoca di fatto la versione di preleva di `ContoConFido`.

Assegnazione dal generale al particolare ?

- Non si può assegnare un oggetto da generale al particolare, es. `ce=cb`
  - `cb` non ha campi e valori corrispondenti ai campi particolari introdotti dalla classe conto con fido
  - non ha senso proiettare un punto dal piano cartesiano X-Y nello spazio, dal momento che non è definita la coordinata z
- Tuttavia, se `cb` ha tipo dinamico `ContoConFido`, si può di fatto cambiare punto di vista ("paio di occhiali") su `cb` in modo da vederlo come `ContoConFido` e quindi accedere a tutte le funzionalità di `ContoConFido`

```
if( cb instanceof ContoConFido )(
ce=(ContoConFido)cb; //casting
ce.nuovoFido(5000);
}
```

- Su una variabile `cb` di classe (tipo statico) `ContoBancario` possono essere richieste sempre e solo le funzionalità della classe cui appartiene
- Se `cb` ha tipo dinamico `ContoConFido`, invocando un metodo ridefinito in `ContoConFido` come preleva/deposita, di fatto si esegue la versione del metodo di `ContoConFido`
- Se `cb` ha tipo dinamico `ContoConFido`, controllabile con `instanceof` è allora possibile cambiare il punto di vista su `cb` (*casting*)

Dynamic binding e polimorfismo

- Il dynamic binding (collegamento dinamico) si riferisce alla proprietà che invocando un metodo su un oggetto come `cb`, dinamicamente possa essere eseguita la versione del metodo definita in: `ContoBancario` oppure `ContoConFido`
- Il termine polimorfismo significa "più forme" ed esprime la proprietà che un oggetto possa appartenere a più tipi
  - con `cb=ce`, l'oggetto `cb` acquisisce un altro tipo (diventa polimorfo)
  - Il polimorfismo di `cb` si può verificare come segue

```
if(cb instanceof ContoBancario ) è TRUE
if(cb instanceof ContoConFido ) è TRUE
```

- dynamic binding e polimorfismo sono le due facce di una stessa medaglia:
  - Il polimorfismo è la causa del dynamic binding

Ereditarietà e ridefinizione dei metodi

- `ContoConFido` ridefinisce i metodi `deposita` e `preleva` già presenti nella `super` classe `ContoBancario`
  - occorre normalmente rispettare la sua intestazione (signature)
  - se cambia qualcosa nell'intestazione (nome del metodo, tipi dei parametri): *overloading* anziché di ridefinizione (*overriding*).
- Perchè funzioni correttamente il dynamic binding/polimorfismo, è necessario osservare l'esatta intestazione

```
@Override // ANNOTAZIONE FACOLTATIVA!!!!
public boolean preleva( doublé quanto ){...}
```

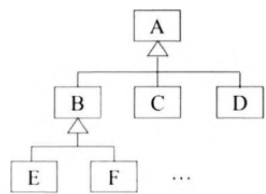
- L'annotazione permette al compilatore di controllare ed eventualmente segnalare problemi

Ereditarietà singola

- In Java ogni classe può essere erede di una sola classe (ereditarietà singola).
- Tutto ciò permette la costruzione di gerarchie di classi secondo una struttura ad albero
  - Ogni classe ha solo un genitore
- Avere una gerarchia accresce la possibilità di polimorfismo

Esempio

- Oggetti di classe E sono anche di classe: B, A
- Ad una variabile di classe A è possibile assegnare un oggetto di qualsiasi sottoclasse: B, C, D, E, F
- Il tipo dinamico di un oggetti di classe A può essere uno qualsiasi delle sottoclassi



Ereditarietà vs composizione

- Riflessione sulla relazione di ereditarietà alla luce del principio di sostituibilità dei tipi

Esempio

- Un oggetto `Linea` (segmento) è caratterizzato da due punti (oggetti di classe `Punto`)
- È corretto definire `Linea` come sottoclasse di `Punto`?

Ereditarietà vs composizione

- Riflessione sulla relazione di ereditarietà alla luce del principio di sostituibilità dei tipi

Esempio

- Un oggetto `Linea` (segmento) è caratterizzato da due punti (oggetti di classe `Punto`)
- È corretto definire `Linea` come sottoclasse di `Punto`?
- No! Rappresenta una forzatura.
- Una `Linea` non è un `Punto`, ma è *composta* (has-a) da punti
- Pertanto la cosa migliore è definire la classe `Linea` come segue:

```
class Linea {
  Punto p1, p2;
}
```

L'antenato Object

- In Java, ogni classe eredita direttamente o indirettamente da `Object` (radice di tutte le gerarchie di classi)
- Quando una classe non specifica la clausola extends, in realtà ammette implicitamente la clausola: `extends Object`
- I metodi seguenti ammettono già un'implementazione in `Object` che necessariamente è generica. Essi vanno

di norma ridefiniti per avere un significato "tagliato su misura" delle nuove classi: • `String toString()` - ritorna lo stato di `this` sotto forma di stringa

- `boolean equals( Object x )` ritorna true se this ed x sono uguali
  - `Object` definisce l'uguaglianza in modo superficiale: due oggetti sono uguali se sono in aliasing, ossia condividono lo stesso riferimento
- `int hashCode()` - ritorna un hash code (numero intero unico) per `this`

Strutture Dati Eterogenee

- Grazie alla ereditarietà implicita da `Object` possiamo dichiarare strutture dati eterogenee come segue:

```
Object[] v = new Object[10];
```

- in `v` possiamo memorizzare oggetti appartenenti a qualsiasi classe
- per scoprire il tipo di un oggetto contenuto in `v` possiamo scrivere

```
if(v[i] isinstanceof String)   ...
```

Recap: modificatori di accesso

- Gli attributi di una classe (campi o metodi) possono avere un modificatore tra
  - `public` se sono esportati a tutti i possibili client
  - `private` se rimangono ad uso esclusivo della classe
  - `protected` se sono esportati solo alle classi eredi
  - (nulla) se devono essere accessibili all'interno dello stesso package (familiarità o amicizia tra classi).
- Attenzione: gli attributi `protected` sono accessibili anche nell'ambito del package di appartenenza.
- Una classe può essere `public` se è esportata per l'uso in altri file, non avere il modificatore `public` se il suo uso è ristretto al `package` (eventualmente anonimo) di appartenenza.
- Una classe può essere `final` se non può essere più estesa da classi eredi.
  - similmente, un metodo `final` non può essere più ridefinito nelle sottoclassi
- In una ridefinizione di metodo è possibile ampliare il suo modificatore ma non restringerlo
- Ad es. nella super classe il metodo potrebbe essere `protected` e nella sotto classe `public`, ma non viceversa.

Esercizi

ContoBancario

- Si implementi la gerarchia di classi `ContoBancario`
- Si implementi una classe `BancaArray` che contenga al suo interno una collezione di conti bancari (possono essere di tipo `ContoBancario` oppure `ContoConFido`)

Contatore

- Si consideri una classe `Contatore` che fornisce l'astrazione di un contatore, ossia una variabile intera che può essere incrementata/decrementata.
- La classe dispone di tre costruttori:

1. quello di default che inizializza a zero il contatore
2. quello normale che imposta il valore iniziale del contatore con il valore di un parametro
3. quello di copia che imposta il contatore dal valore di un altro contatore. Per semplicità il campo valore è dichiarato protected (esportato cioè alle classi eredi).

- Si implementi una seconda classe `ContatoreModulare` che erediti
  - Un contatore con modulo 10 è un contatore che assume tutti i valori da 0-9. Una volta raggiunto il valore 9, ritorna nuovamente a 0.

Presentation agenda

Value vs Reference

Ereditarietà, dynamic binding, polimorfismo

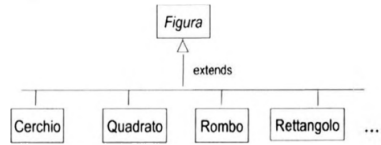
Classi Astratte e Interfaccia

Tipi di dati astratti

Collection framework

Una gerarchia di classi per figure geometriche piane

- Si considerano le comuni figure piane:
  - cerchio, quadrato, rombo, trapezio ...
- Si vuole organizzare le figure in modo da facilitarne l'utilizzo nelle applicazioni
  - tutte posseggono almeno una dimensione,
    - il raggio per il cerchio
    - il lato per il quadrato o il rombo
    - la base e l'altezza per un rettangolo
- Per "imparentare" le figure si può concepire una classe base `Figura` che poi ogni figura particolare può estendere e specializzare
  - in Figura si può introdurre una dimensione (double) e i metodi che certamente hanno senso su tutte le figure.



Discussione

- Identificare una gerarchia di classi come quella di cui si sta discutendo ha una cruciale importanza
  - Si può introdurre nella classe base (`Figura`) tutti quegli elementi (attributi e metodi) comuni a qualunque erede.
    - In questo modo si evitano ridondanze e si garantisce ad ogni classe derivata di possedere i "connotati" di appartenenza ad una stessa "famiglia".
- Si rifletta ora che prevedendo una dimensione (cioè un lato) nella classe `Figura`, il suo concreto significato non è chiaro
  - per un cerchio si tratterà del suo raggio
  - per un quadrato del suo lato
  - per un rettangolo la sua base
- Metodi come `perimetro()` ed `area()` previsti in `Figura` non si possono dettagliare in quanto manca l'informazione su come interpretare la figura
- Si dice che una classe come Figura è astratta (abstract) proprio perchè ancora incompleta
  - Spetterà poi alle classi eredi concretizzare tutti quegli aspetti previsti in `Figura` ma al momento astratti

Implementazione classe Figura

```
public abstract class Figura {
    private double dimensione;
    public Figura( double dim ){
if( dim<=0 )
    throw new IllegalArgumentException();
this.dimensione=dim;
    }

    protected getDimensione(){ return dimensione;}
    public abstract double perimetro();
    public abstract double area();
}
```

Discussione II

- Una classe astratta come `Figura` non è istanziabile.

Discussione II

- Una classe astratta come `Figura` non è istanziabile.
  - Allora a che ... serve ?

Discussione II

- Una classe astratta come `Figura` non è istanziabile.
  - Allora a che ... serve ?
  - Serve come base per progettare classi eredi!
- Per definire una classe astratta si deve premettere al nome class la keyword `abstract`
- In una classe astratta uno o più metodi sono di norma astratti.
  - Una classe erede è concreta se implementa (ne fornisce cioè il corpo) tutti i metodi abstract.
    - Se qualche metodo rimane ancora astratto, anche la classe erede è astratta e spetta ad un ulteriore erede implementare i rimanenti metodi abstract etc.
- Si nota che in una classe astratta possono essere presenti campi dati (es. dimensione) e metodi concreti.
  - Ad esempio `getDimensione()`, utile solo per le classi eredi (esportazione protected), è concreto.

Implementazione classe Cerchio

```
public class Cerchio extends Figura{
    public Cerchio( double raggio )( super(raggio);}
    public Cerchio( Cerchio c ){ super(c.getDimensione());}
    public double getRaggio(){ return getDimensione();}

    public double perimetro(){ return 2*Math.PI * getDimensione(); }//perimetro
    public double area(){
double r=getDimensione();
return PI*r;
    }
    public String toString(){
return "Cerchio: raggio="+getDimensione();
    }//toString
   public boolean equals(Object other){...}
}//Cerchio
```

- Essendo privato il campo dimensione di `Figura`, si è fatto ricorso ai metodi `getDimensione()`/`setDimensione()` per accedervi da dentro `Cerchio`.
- Il metodo `equals()` necessariamente è peculiare di ogni classe erede, e per questa ragione non è stato previsto in `Figura`.
  - Similmente per il metodo `toString()`
  - In altre situazioni può essere invece conveniente anticipare nella super classe una implementazione dei metodi `equals()`, `hashCode()` e `toString()`

Esercizi

- Si implementi una classe `Rettangolo`
- Si implementi una classe `Utility` che contenga soltanto metodi statici. Fornire l'implementazione della seguente funzione:
  - `areaMassima`: riceve in ingresso una collezione di figure e restituisce quella con l'area massima.

Una classe astratta per il problema dell'ordinamento

- Di seguito sivaluta la possibilità di risolvere il problema dell'ordinamento di un array di oggetti
- Occorre avere una classe che fornisce un metodo di ordinamento che si fonda su un criterio di confronto da specializzare di caso in caso.
  - In fondo, la logica dell'ordinamento è sempre la stessa, indipendentemente dalla tipologia degli oggetti
    - Occorre specializzare il concetto di minore/maggiore

```
public abstract class Sortable{
   protected abstract int compareTo( Sortable x );
   public static void sort( Sortable []v ){
  for( int j=v.length-1; j>0; j - ){
      int iMax=0;
      for( int i=0; i<=j; i++ )
  if( v[i].compareTo(v[iMax))>0 ) iMax=i;
      //scambia
      Sortable park=v[j];
      v[j]=v[iMax];
      v[iMax]=park;
  }//for
   }//sort
}//Sortable
```

- Il metodo `compareTo` deve sostituire:
  - 0 se l'istanza passata in input è uguale a `this`
  - un valore <0 se `this` è minore dell'istanza passata in input
  - un valore >0 se `this` è maggiore dell'istanza passata in input

Esercizi

Es. 1

- Si definisca una classe `Intero`, erede di `Sortable`, che memorizzi al suo interno il valore di un numero intero, sulla base del quale dovrà essere eseguito l'ordinamento
- Testare il funzionamento della nuova classe

Es. 2

- Inglobare la classe `Sortable` nel progetto relative alla class `Razionale`
- Testare il funzionamento

Limiti dell'approccio

- L'approccio non è applicabile se una classe i cui oggetti si vogliono ordinare, è già legata in una gerarchia di ereditarietà e dunque non può estendere `Sortable`
  - volendo ordinare oggetti di classe `Impiegato extends Persona`, dunque non possiamo avere `Impiegato extends Persona, Sortable`

Limiti dell'approccio

- L'approccio non è applicabile se una classe i cui oggetti si vogliono ordinare, è già legata in una gerarchia di ereditarietà e dunque non può estendere `Sortable`
  - volendo ordinare oggetti di classe `Impiegato extends Persona`, dunque non possiamo avere `Impiegato extends Persona, Sortable`

Polimorfismo

- Operazioni lecite? —>

```
class Impiegato implements Persona, Sortable {....}
```

```
public static void main(String[] args) {
  Persona p = new Impiegato(....);
  Sortable s = new Impiegato(...);
}
```

Il concetto di interfaccia

- Rappresentano un meccanismo per simulare l'eredità multipla: le interfacce
  - Una classe può estendere una sola classe ma può implementare zero, una o più interfacce
- Un'interfaccia (`interface`) è una raccolta di intestazioni (segnature) di metodi
  - Le segnature di metodi sono definizioni astratte pur senza il modificatore `abstract`
  - Una classe che implementi un'interfaccia deve fornire un'implementazione di tutti i metodi definiti nell'interfaccia, altrimenti la classe è da riternersi astratta.
- Ammette definizioni di attributi costanti e tipi innestati
- Un'interfaccia, cosi come una classe astratta, non è istanziabile.

A proposito di Sortable

- Per massima generalità `compareTo()` lavora su `Object`
  - `compareTo(x)` restituisce un valore:
    - `<0, ==0, >0` se l'oggetto `this` è rispettivamente minore, uguale o maggiore di `x`.

```
public interface Comparable{
public int compareTo( Object x );
}//Comparable
```

Razionali comparabili

```
  public class Razionale implements Comparable{
  //... come prima
  public int compareTo( Object x ){
      Razionale r=(Razionale)x;
      int mcm=(this.denominatore*r.denominatore)/mcd(this.denominatore, r.denominatore);
      int n1=(mcm/this.denominatore)‘ this.numeratore;
      int n2=(mcm/r.denominatore)*r.numeratore;
      if( n1<n2 ) return -1;
      ifj n1>n2 ) return 1;
      return 0;
  }//compareTo
}//Razionale
```

- `Razionale` estende (implicitamente) Object, discende che i razionali sono anche di tipo `Object`
- `Razionale` implementa `Comparable`, deriva che gli oggetti razionali sono anche comparabili, ossia di tipo `Comparable` (aumento del polimorfismo).
  - Un array di `Comparable` è dunque un array di oggetti sui quali è definito il criterio di confronto.

La classe di utilità Array

```
  public final class Array{//versione completa fornita a parte
  private Array(){}
  public static void selectionSort( Comparable []v ){ ... }
  public static void bubbleSort( Comparable []v ){ .... }
  public static int ricercaBinaria( Comparable []v, Comparable x ){...}
}
```

Discussione

- L'uso dell'interfaccia `Comparable` rende possibile approntare una classe di utilità come Array che esporta i più comuni algoritmi di ordinamento e ricerca (lineare e binaria).
  - Diverse varianti sono disponibili di uno stesso metodo (overloading):
    - es. litisconsortile , di Comparable
    - c'è una versione che accetta un array di int e un'altra che accetta un array di double
- Questo modo di opeare, come si vedrà nel seguito, è ampiamento sfruttato dalla libreria di Java (API)
- Per avvalersi di un metodo qualsiasi di ordinamento di Array, è

sufficiente che una classe applicativa implementi `Comparable`

- Quando una classe implementa `Comparable`, si dice che i suoi oggetti dispongono dell'ordinamento naturale
- L'approccio basato sull'interfaccia lascia libera una classe di ereditare da una super classe
  - Non sussistono più i limiti riscontrati con il metodo basato sulla classe astratta `Sortable`
- Le interfacce possono essere costruite anche per estensione (`extends`)
  - Se l'interfaccia `I2` estende `I1`, allora banalmente in `I2` si ritrovano tutte le intestazioni di metodi di `I1` più quelle previste da `I2`

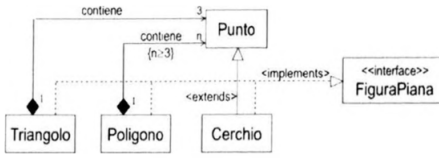
Regole di un buon progetto di una classe Java

- Alla luce delle conoscenze sin qui acquisite, si può dire che il progetto di una classe, per generalità, dovrebbe:
  - prevedere il metodo `boolean equals(Object x)`
  - prevedere il metodo `String toString()`
  - prevedere il metodo `int hashcode()` che ritorna un intero identificativo unico dell'oggetto
    - Se due oggetti sono uguali secondo `equals`, allora il loro hashcode deve essere uguale
    - Tuttavia oggetti non uguali possono avere lo stesso valore di hashcode
  - Per definire i metodi equals() e hashCode() occorre prestare ai campi (di norma immutabili) che identificano un oggetto
    - per una persona potrebbero essere cognome e nome o il campo codice fiscale
    - per uno studente la matricola etc
- Implementare l'interfaccia `Comparable` e dunque il metodo `compareTo`, se si prevede che gli oggetti debbano essere assoggettati ad ordinamento o comunque a confronti (es. per ragioni di ricerca)

Un altro esempio di uso delle interfacce

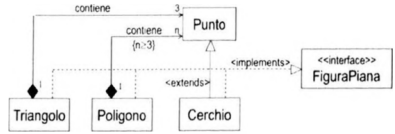
- La relazione con il rombo indica che `Triangolo` contiene 3 (molteplicità della relazione) punti
- La linea tratteggiata terminante con una freccia bianca indica che Triangolo implementa l'interfaccia `FiguraPiana`
- `Cerchio` estende `Punto` e implementa `FiguraPiana`.
- Ovviamente, un'interfaccia può far parte di un package esplicito ed essere raccolta in un file
- Essa va compilata come le classiPonendo FiguraPiana in

```
public interface FiguraPiana{
double perimetro();
double area();
}//FiguraPiana
public class Triangolo implements FiguraPiana {
public double perimetro(){...}
public double area(){...}
}//Triangolo
```



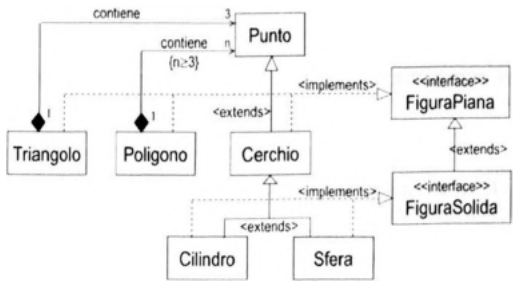
Discussione

- Un'interfaccia consente di accomunare classi che diversamente resterebbero isolate
- Consideriamo una semplice gerarchia: dalla classe `Punto` si deriva `Cerchio`, che in più aggiunge il raggio
  - in precedenza abbiamo definito una classe `Triangolo` che contiene tre punti
  - `Triangolo` e `Poligono` non estendono `Punto` perché cotengono punti
- Le tre classi, `Cerchio`, `Poligono` e `Triangolo` non condividono niente
  - tuttavia, potrebbe essere conveniente introdurre una interfaccia `FiguraPiana` con i due metodi necessari al calcolo di perimetro e area
  - si impone quindi alla classe `Cerchio`, `Triangolo` e `Poligono` di implementare questa interfaccia comune, appunto `FiguraPiana`
    - in questo modo possiamo memorizzarle in una stessa struttura dati



Discussione

- Il discorso può proseguire ulteriormente:
  - Da `Cerchio` si può derivare `Sfera` che è una figura solida.
  - A questo punto si potrebbe definire un'interfaccia `FiguraSolida` che extends `FiguraPiana` ed aggiunge
    - metodi come double `areaLaterale()` e `double volume()`



Esercizi

- Implemetare la gerarchia della slide precedente introducendo anche la `FiguraSolida`
- Programmare altre classi eredi di Figura come Quadrato, Rombo, Triangolo,TrapeziolsoscelE, etc. La

classe Triangolo potrebbe esportare un metodo per conoscere il tipo di triangolo etc.

- Programmare una classe Cono che estende Cerchio e implementa l'interfaccia FiguraSolida.

Presentation agenda

Value vs Reference

Ereditarietà, dynamic binding, polimorfismo

Classi Astratte e Interfaccia

Tipi di dati astratti

Collection framework

Tipi di dati astratti

- Spesso le applicazioni utilizzano dati strutturati (aggregati) che si caratterizzano per le operazioni che si debbono eseguire sui dati e non per il modo in cui questi aggregati sono rappresentati in memoria.
- Tutto ciò introduce il concetto di tipo di dati astratto (ADT o abstract data type) che in Java è esprimibile in modo naturale con una interfaccia o una classe astratta
  - si tratta di un pacchetto di metodi (contratto) specificati unicamente mediante le loro intestazioni.
  - un ADT è poi concretizzabile (implementabile) in diversi modi,

Esempio

- Si vuole realizzare una nozione di array (vector) "più comoda" per le applicazioni, rispetto all'array nativo di Java.
  - gli array nativi sono strutture dati compatte e statiche e tendono ad introdurre problemi quando si vuole aggiungere un elemento e l'array è pieno, o quando si vuole eliminare un elemento senza creare buchi
- Un vector è pensato scalare automaticamente di dimensione ogni volta che serve, e farsi carico trasparentemente delle eventuali operazioni di spostamento di elementi a seguito di inserimenti o rimozioni.
- In quanto segue si definisce un ADT Vector mediante un'interfaccia
  - gli sono assunti `Object` per generalità.
  - successivamente, l'utilizzo del meccanismo dei generici di Java consentirà di migliorare in flessibilità e sicurezza la definizione ed uso dei vector.

ADT Vector

L'ADT Vector:

```
package poo.util;
public interface Vector{
public int size();
public int indexOf( Object elem );
public boolean contains( Object elem );
public Object get( int indice );
public Object set( int indice, Object elem );
public void add( Object elem );
public void add( int indice, Object elem );
public void remove( Object elem );
public Object remove( int indice );
public void clear();
public boolean isEmpty();
public Vector subVectorj int da, int a );
}
```

Semantica Operazioni I

- int sizeQ
  - ritorna il numero di elementi presenti nel vettore. Gli elementi del vettore, similmente agli array, hanno indici 0 e size()-1
- int indexOf(Object elem)
  - ritorna l'indice della prima occorrenza di elem nel vettore, o -1 se elem non è presente. Si basa sul metodo equals degli elementi.
- boolean containsf Object elem)
  - ritorna true se elem è presente almeno uno volta nel vettore, false altrimenti. Si basa sul metodo equals degli elementi.
- Object get(int indice)
  - ritorna l'elemento alla posizione indice del vettore. Sostituisce la notazione vindice] degli array nativi. Solleva un'eccezione IndexOutOfBoundsException se indice non è compreso tra 0 e size()-1
- Object set(int indice, Object elem)
  - suppone il valore di indice compreso tra 0 e size()-1. Sostituisce l'elemento alla posizione indice con elem, e ritorna il precedente elemento. Solleva un'eccezione IndexOutOfBoundsException se l'indice non è valido

Semantica Operazioni II

- void add(Object elem)
  - aggiunge elem come ultimo elemento del vector, espandendo la struttura se necessario.
- void add(int indice, Object elem)
  - aggiunge elem alla posizione indice, spostando preliminarmente di un posto a destra tutti gli elementi da indice in poi. Espande la struttura se necessario. Solleva una eccezione IndexOutOfBoundsException se indice non è compreso tra 0 e sizeQ
- void removef Object elem)
  - elimina, se esiste, la prima occorrenza di elem dal vector
- Object removej int indice)
  - elimina l'elemento alla posizione indice, e lo ritorna. Solleva un'eccezione IndexOutOfBoundsException se indice non è compreso tra 0 e size()-1
- void clearQ
  - svuota il vector. Dopo l'operazione size() vale 0
- boolean isEmptyQ
  - ritorna true se size()==0
- Vector subVectorj int da, int a)
  - crea un nuovo vector e vi copia gli elementi dalla posizione da alla posizione a (esclusa) di this. Sollevaun'eccezione se gli indici non sono validi: da deve essere in [0,size()-1], a in [O.sizeQ]

Esercizio

- Fornire una implementazione di `Vector` basata su array che rispetti le seguenti specifiche:
  - Deve essere presente un attributo `size`, il cui valore indica il primo slot libero (se esiste) dell'array
    - il metodo `add(elem)` deve inserire nella posizione puntata da size
  - Le espansioni/contrazioni dell'array sono curate rispettivamente da metodi ausiliari privati (e.g., `espandi`, `contrai`)
    - si espande quando il valore di `size` eguaglia quello della lunghezza dell'array
    - si contrae quando il valore di `size` scende oltre metà della lunghezza dell'array
  - L'aggiunta (resp. rimozione) di un elemento intermedio comporta lo scorrimento a destra (resp. sinistra) del contenuto dell'array
  - La remove la vecchia ultima posizione dell'array viene posta a `null` (per favorire il garbage collector)
  - Il metodo `hashCode` utilizza una tecnica canonica:
    - si combinano gli hash code degli elementi componenti utilizzando un fattore di *shuffling*

Discussione

- Poiché gli elementi di un vector sono Object, tutti i tipi di oggetti, istanze cioè di una qualsiasi classe, posson essere memorizzati
- `Vector` è una struttura dati potenzialmente eterogenea
  - possono essere inseriti oggetti `String` unitamente ad oggetti razionali, oggetti punti etc
  - Lavorare con un tale tipo di struttura non pone problemi sino a che si richiedono operazioni comuni a tutte le classi: `toString`, `equals()`
- Per applicare metodi specifici di un particolare tipo di oggetto, occorre identificare il suo tipo dinamico (con `instanceof`) e quindi (mediante casting) applicare il punto di vista della relativa classe
- Tuttavia, è da notare come nella maggior parte dei casi la classe `Vector` verrà utilizzata per memorizzare oggetti appartenenti alla stessa classe, i.e., omogenei

Un Vector generico e parametrico

- A partire dalla versione 5 Java, ha introdotto il meccanismo dei generici
- I generici offrono la possibilità di programmare una classe/interfaccia (o anche singoli metodi) in veste generica un uno o più tipi (parametri tipi formali)
  - ADT Vector diventa più flessibile e sicuro se viene riprogettato in veste generica con un tipo parametrico `T`
- La notazione `Vector<T>` significa che la struttura dati è composta di elementi tutti di uno stesso tipo generico `T`
  - `T` può essere sostituito con una qualsiasi classe Java

```
public interface Vector<T> {
...
}
```

```
Vector<Integer> v =
    new ArrayVector<Integer>();
Vector<String> w =
    new ArrayVector<String>();
```

Discussione

- Con una tale organizzazione si ottenono diversi benefici:
  - il compilatore garantisce che in `v` non si possano inserire elementi che non siano oggetti `Integer` (omogeneità)
- la parametricità garantisce che quando si preleva da `w` l'oggetto sarà sicuramente una `String`, quindi non serve più il casting da `Object`

Classi wrapper dei tipi primitivi

- Poiché il tipo parametro formale `T` di una classe generica come `ArrayVector<T>` denota una qualsiasi classe Java
  - va da sé che il meccanismo dei generici non permette di utilizzare direttamente i tipi primitivi (che non sono classi).
  - non si può scrivere `ArrayVector<int>` ma solo `ArrayVector<Inteqer>`
  - Per generalità il linguaggio introduce alcune classi predefinite che sono associate ai tipi primitivi (classi wrapper):
    - int -> Integer, a byte->Byte, short ->Short, a long >Long, a floatloat, a double->Double, a char->Character, a boolean->Boolean.
    - Le classi numeriche (e.g., Integer,Double etc.) sono eredi della classe astratta Number.
  - Per ovvie ragioni, un oggetto di una classe wrapper è immutabile perché rappresenta una costante di un tipo primitivo sebbene sotto forma di oggetto
  - Le classi wrapper offrono metodi e attributi di utilità generale.
    - tute sono di tipo `Comparable` e sono provviste di: `toString()`, `equals()`, ...
  - Per semplificare la vita al programmatore Java si occupa delle opearzioni di boxing/unboxing
    - Le seguenti istruzioni sono pertanto lecite:

```
Vector<Integer> v = new ArrayVector<Integer>();
v.add(5); //boxing
int x=v.get(0); //unboxing
```

Vector<T> generico e parametrico

- La programmazione di una classe con tipi parametrici è vincolata da alcune semplici regole:
  - per il nome del tipo parametro formale utilizzare semplici lettere (e.g., T, E, K)
  - il parametro formale rappresenta qualsiasi classe senza informazioni specifiche
- Non è possibile istanziare un oggetto di tipo T tramite l'operatore new
  - nemmeno la creazione di un array di tipo T è consentita,
    - sebbene la difficoltà può essere aggirata creando un array di Object e poi castizzando tale array al tipo (T[])
- Oggetti di tipo T possono essere ricevuti/restituiti
- Su oggetti di tipo T è possibile richiamare i metodi standard degli oggetti Java (i.e., quelli ereditati da Object)

Ridefinizione della classe Vector

```
public interface Vector<T>{
    public int size();
    public int indexOf( T elem );
    public boolean contains( T elem );
    public T get( int indice );
    public T set( int indice, T elem );
    public void add( T elem );
    public void add( int indice, T elem );
    public void remove( T elem );
    public T remove( int indice );
    public void clear();
   public boolean isEmpty();
   public Vector<T> subVector( int da, int a );
}//Ve ctor
```

Esercizio

- Implementare l'interfaccia =Vector<T>=K

Discussione

- La nuova implementazione rinuncia ai metodi di ausilio `espandi()` e `contrai()` in quanto ottiene l'equivalente funzionalità mediante il metodo `java.util.Arrays.copyOf( array_source, dim )` che crea e ritorna un array dello stesso tipo di $\text{array}_{\text{source}}$, di capacità dim (maggiore/minore di array„source.length), e copia gli elementi di $\text{array}_{\text{source}}$ nel nuovo array nel quale le posizioni vacanti sono poste a null

Presentation agenda

Value vs Reference

Ereditarietà, dynamic binding, polimorfismo

Classi Astratte e Interfaccia

Tipi di dati astratti

Collection framework