Programmazione ad Oggetti

Ereditarietà e Interfacce

Mercoledi, Aprile 03, 2023

Claudio Menghi,

Contents

1 Ripasso degli elementi del linguaggio Java

1.1 Ereditarietà

Nella programmazione ad oggetti l'*ereditarietà* è un meccanismo tramite il quale è possibile definire una classe a partire da un'altra e riutilizzare gli attributi e i metodi già definiti. I costruttori *non* sono ereditati, anche se è possibile chiamare i costruttori del padre nel figlio.
È detta classe figlio la classe che eredita, classe padre la classe ereditata. La classe figlio ha accesso a tutti gli attributi e i metodi della classe padre a seconda della loro visibilità. Tutti gli attributi e i metodi `public` e `protected` sono visibili quelli di `default` solo se padre e figlio risiedono nello stesso package.
Con l'eccezione di `Object`, che non ha eredita alcuna classe, ogni classe ha una unica classe padre (ereditarietá singola). Se una classe non specifica alcuna classe padre, questa estende implicitamente la classe `Object`.
L'idea dell'ereditarietà è molto semplice: quando vuoi creare una nuova classe A ed esiste già una classe B che contiene del codice che desideri puoi derivare A da B. In questo caso puoi riusare i campi e i metodi della classe esistente senza doverli riscrivere.
Che cosa è possibile fare con l'ereditarietà:

- i campi ereditati possono essere usati direttamente, esattamente come ogni altro campo;
- è possibile dichiarare attributi nella sottoclasse che non sono presenti nella classe padre;
- è possibile utilizzare i metodi ereditati
- è possibile scrivere un metodo nella sottoclasse con la stessa signature di quello della superclasse, mascherandolo
- è possibile srivere un metodo statico nella sottoclasse con la stessa signature di quello della super-classe, mascherandolo
- è possibile dichiarare nuovi metodi nella sottoclasse che non sono presenti nella super-classe
- è possibile scrivere nella sotto-classe un costruttore che invoca un costruttore della superclasse usando la keyword `super`

1.2 Overriding dei metodi

La classe figlio può *ridefinire* (*override*) metodi visibili nella classe padre **N.B.: Si parla di overriding solo se vi è visibilità del metodo del padre nella classe figlio**. La signature del metodo ridefinito deve corrispondere alla signature del metodo del padre.
Nella ridefinizione è possibile cambiare la visibilità dei metodi e il tipo di ritorno. La nuova visibilità deve essere maggiore (secondo la scala `private` < `default` < `protected` < `public`), mentre il tipo di ritorno deve essere un sottotipo del metodo originale ovvero *covariante*[1].

1.3 Overloading dei metodi

L'overloading consente a una classe di avere due o più metodi con lo stesso nome. Affinchè due o più metodi con lo stesso nome possano essere nella stessa classe è necessario che differiscano almento in almeno uno tra:

- numero di parametri
- tipo dei parametri
- sequenza del tipo dei parametri

[1] Vedremo la definizione di covarianza più avanti nel corso

1.4 Polimorfismo

Si definisce *polimorfismo* il fatto che un oggetto può assumere diverse “forme" ovvero, è possibile assegnare a una variabile di tipo T un oggetto di tipo S dove S è una sottoclasse di T.
Per questo motivo in Java si distingue tra tipo statico e tipo dinamico.

- **tipo statico** Equivale al tipo presente nella dichiarazione della variabile. È unico per tutta la vita della variabile e disponibile a compile-time (basta guardarne la dichiarazione)
- **tipo dinamico**: Equivale al tipo dell'istanza effettiva associata alla variabile (o coincide col tipo statico o è un sottotipo di quest'ultimo). Può cambiare durante l'esecuzione del programma.

In Java è possibile assegnare a una variabile di classe T un'istanza di classe S dove S eredita T. Consideriamo le seguenti istruzioni

```
Automobile myCar=new AutomobileElettrica();
```

Il tipo statico di `myCar` è `Automobile`, il suo tipo dinamico è `AutomobileElettrica`.

1.5 Binding

Si definisce con *binding* l'associazione della chiamata di un determinato *metodo* all'effettivo codice associato al metodo.

A compile time, il compilatore non genera il codice per eseguire il metodo, ma il codice per cercare l'*implementazione* corretta in base al tipo *dinamico* dell'oggetto su cui il metodo è invocato.
Per effettuare il binding Java utilizza il seguente algoritmo. Ipotizziamo di chiamare un metodo `foo` sull'oggetto `o`

```
O o = ...
o.foo(...);
```

1. Per prima cosa si trova il metodo all'interno della classe `O` tipo statico di `o` che abbia la segnatura corrispondente ai tipi statici degli argomenti della chiamata (**binding statico**).
2. A questo punto si guarda al tipo dinamico di `o`: se `o` è un'istanza di un sottotipo di `O`, `S`, si deve controllare se `S` ridefinisca o meno `foo`. Se è ridefinito utilizzo l'implementazione del sottotipo, altrimenti quella di `O` (**binding dinamico**).

Nei metodi ridefiniti la visibilità e il tipo di ritorno possono essere cambiati: la visibilità può solo essere aumentata ed è il tipo di ritorno può solo essere cambiato in un suo sottotipo.

2 Esercizi

2.1 Scacchiera

Esercizio 1: Si implmenti il gioco degli scacchi

- Si ha una griglia 8x8 denominata Scacchiera
- Ogni elemento della griglia si chiama Casella
- In ogni casella ci può essere al più un Pezzo
- I pezzi possono essere: Torre, Cavallo, Alfiere, Regina, Re, ognuno con differenti capacità di movimento (si ignori il Pedone per il momento).
- La scacchiera viene creata con dei pezzi dentro le caselle posizionati opportunamente.
- Una casella può essere vuota o avere un pezzo al suo interno.
- Un pezzo può appartenere al giocatore Bianco o al giocatore Nero e può spostarsi da una casella ad unaltra secondo determinate regole dipendenti dal pezzo.
- Un pezzo può muoversi solo verso una casella vuota od una casella occupata da un pezzo avversario. In questo secondo caso il pezzo avversario viene rimosso. I movimenti consentono di spostarsi di al più una casella in orizzontale, verticale o obliquo.

Il primo passo da eseguire è analizzare il problema ed identificare le entità (ovvero delle classi candidate). In questo caso, abbiamo `Scacchiera`, `Casella`, `Pezzo`, ...

Colori

Iniziamo ad analizzare un sottoproblema: modelliamo il colore delle pedine.

Esercizio 2: Le pedine possono avere 2 colori: *nero* e *bianco*

- come modelliamo i colori?
  - i colori sono modellabili con un set *predefinito* di costanti: `BLACK` and `WHITE`

```
package scacchi.scacchiera;

// Le pedine possono essere di colore bianco o nero: un set definito a priori E NON
    modificabile di valori
public enum Color {
      WHITE, BLACK
}
```

2.1.1 Casella

Un nuovo sottoproblema: modelliamo la casella.

Esercizio 3: Analizziamo le caratteristiche di una casella:

- ogni casella ha una coordinata
- ogni casella può avere un pezzo al suo interno
- deve essere possibile aggiungere un pezzo all'interno della casella
- deve essere possibile controllare se all'interno della casella è presente un pezzo

Leggendo la specifica identifichiamo un nuovo elemento: `Coordinata`:

- è una classe mutabile?
  - sembra ragionevole implementare una `Coordinata` per mezzo di una classe immutabile
- Quali `funzionalità` offre una coordinata?
  - `int getX()` ritorna la coordinata X della Casella
  - `int getY()` ritorna la coordinata Y della Casella (assumiamo che la casella 0,0 sia in alto a sinistra e la casella 8,8 sia in basso a destra)
- Come memorizziamo lo stato di una coordinata?
  - memorizziamo lo stato per mezzo di due interi: x,y entrambi final
- Come deve essere il costruttore di un oggetto di tipo `Coordinata`?
  - `Coordinata(int x, int y)`

```
package scacchi.scacchiera;

public class Coordinata {

	private final int x;
	private final int y;

	public Coordinata(int x, int y){
		this.x=x;
		this.y=y;
	}

	public int getX() {
		return x;
	}

	public int getY() {
		return y;
	}
}
```

Analizziamo ora una `Casella`.

- è una classe mutabile?
  - No, deve essere possibile aggiungere/rimuovere un pezzo da una casella
- Quali `funzionalità` offre una coordinata?
  - `boolean isEmpty()` ritorna true se nella casella non è presente un pezzo
  - `Pezzo getPezzo()` ritorna il pezzo contenuto sulla casella, null se non sono presenti pezzi sulla casella.
  - `void setPezzo(Pezzo pezzo)` pone il `Pezzo` pezzo nella casella, se un pezzo era già presente è rimosso dal gioco.
  - `Pezzo unsetPezzo()` rimuove il pezzo dalla casella e ritorna il pezzo rimosso
- Come memorizziamo lo stato di una `Casella`?
  - `Pezzo` contiene il pezzo contenuto nella casella, `null` se nessun pezzo è contenuto
- Come deve essere il costruttore di un oggetto di tipo `Casella`?
  - `Casella(int x, int y)` dove x e y sono le coordinate della casella

```java
package scacchi.scacchiera;
// casella estende coordinata

import scacchi.scacchiera.pezzi.Pezzo;

public class Casella {

	private Pezzo pezzo;
	private final Coordinata coordinata;

	public Casella(Coordinata coordinata) {
		// chiama il costruttore di coordinata
		setPezzo(null);
		this.coordinata = coordinata;
	}

	public Pezzo getPezzo() {
		return pezzo;
	}

	public void setPezzo(Pezzo pezzo) {
		this.pezzo = pezzo;
	}

	public boolean isEmpty() {
		if (this.pezzo == null) {
			return true;
		}
		return false;
	}

	public Pezzo unsetPezzo() {
		Pezzo pieceToBeReturned = this.pezzo;
```

```
            this.pezzo = null;
            return pieceToBeReturned;
      }

      @Override
      public String toString() {
            if (this.getPezzo() != null) {
                  return this.getPezzo().toString();
            } else {
                  return "     ";
            }

      }

      public Coordinata getCoordinata() {
            return coordinata;
      }

}
```

2.1.2 Pezzi

Esercizio 4: I pezzi possono essere: Torre, Cavallo, Alfiere, Regina, Re, ognuno con differenti capacità di movimento (si assuma che il Pedone possa effettuare solo movimenti di una casella).

- Quale relazione esiste fra gli elementi da modellare?
  Nel nostro caso specifico un Torre ***è un*** *pezzo, un Cavallo* ***è un*** *pezzo, una Re* ***è un*** *pezzo.... La relazione* ***è un*** *è esattamente il tipo di relazione che è possibile modellare con la gerarchia.*
- Qual'è la super-classe e qualè la sottoclasse?
  La super-classe è l'oggetto che contiene gli attributi che sono in comune. Nel nostro caso Pezzo è la super classe; Torre, Cavallo, Alfiere... sono sotto-classi.
- Come traduciamo questa relazione in Java? Usiamo una classe, una classe astratta o un interfaccia? *Se deve essere possibile istanziare un elemento con tipo uguale alla super-classe utilizziamo una classe "normale". In caso constrario, un interfaccia fornisce uno scheletro dell'oggetto ma nessuna implementazione. Una classe astratta al contrario può contenere delle parti implementate Nel caso specifico non deve essere possibile istanziare un oggetto di tipo pezzo ma solo di uno dei suoi sottotipi. Quindi optiamo o per una classe astratta o per un interfaccia. Nel caso specifico utilizzeremo una classe astratta e ne vedremo dopo le motivazioni*

Come sempre è necessario pensare in termini di funzionalità e di stato.

- Il pezzo deve fornire le seguenti funzionalità:
  - `Color getColore()` ritorna il colore del pezzo
  - `Casella getCasella()` ritorna la casella corrente del pezzo
  - `void setCasella(Casella casella)` cambia la casella del pezzo
  - `boolean mossaValida(Casella casella)` ritorna `true` se la casella è una destinazione valida per il pezzo (la mossa è consentita dalle regole di movimento del pezzo). Che cosa succede se la casella è null? che cosa succede se la casella non

- come memorizziamo lo stato di pezzo? è necessario tenere traccia di:
  - `Color color` che non deve cambiare dopo l'inizializzazione, quindi è final
  - `Casella casella` contiene la casella corrente del pezzo
- come deve essere il costruttore di `Casella`
  - `public Pezzo(Casella casella, Color color)` quando un `Pezzo` viene creata gli viene associato un colore e una casella iniziale

```
package scacchi.scacchiera.pezzi;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Scacchiera;

public abstract class Pezzo {

	private final Color color;
	private Casella casella;

	public Pezzo(Casella casella, Color color){
		this.setCasella(casella);
		this.color=color;
	}

	public Color getColor() {
		return color;
	}

	public Casella getCasella() {
		return casella;
	}

	public void setCasella(Casella casella) {
		this.casella = casella;
	}
	public abstract boolean mossaValida(Scacchiera scacchiera, Casella
	    destinazione);
}
```

Re La classe `Re` estende la classe Pezzo implementando la logica di movimento della Pedina `Re`. In particolare, saranno valide per il `Re` alcune mosse non valide per il cavallo etc.

```
package scacchi.scacchiera.pezzi;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Scacchiera;

public class Re extends Pezzo {

	public Re(Casella casella, Color color) {
		super(casella, color);
```

```
	}

	@Override
	public boolean mossaValida(Scacchiera scacchiera, Casella casellaFinale) {
		// se la casella finale non \'e nulla
		if (casellaFinale != null) {
			// se la casella non \'e vuota e \'e presente un Pezzo dello stesso
			// colore
			if (!casellaFinale.isEmpty()
					&& casellaFinale.getPezzo().getColor() ==
					    this.getColor()) {
				return false;
			}
			// ritorna true se la distanza tra la cella final e iniziale \'e di
			// una cella in orizzontale e in verticale
			return Math.abs(this.getCasella().getCoordinata().getX() -
			    casellaFinale.getCoordinata().getX()) <= 1
					&& Math.abs(this.getCasella().getCoordinata().getY() -
					    casellaFinale.getCoordinata().getY()) <= 1;

		}
		return false;
	}

	@Override
	public String toString() {
		return "R ["+this.getColor()+"]";
	}
}
```

Alfiere

```
package scacchi.scacchiera.pezzi;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Coordinata;
import scacchi.scacchiera.Scacchiera;

public class Alfiere extends Pezzo {

	public Alfiere(Casella casella, Color color) {
		super(casella, color);
	}

	@Override
	public boolean mossaValida(Scacchiera scacchiera, Casella destinazione) {
		if (destinazione != null) {
			// se la casella destinazione non \'e vuota e \'e presente un Pezzo
			    dello stesso
			// colore ritorna falso
			if (!destinazione.isEmpty()
					&& destinazione.getPezzo().getColor() ==
					    this.getColor()) {
```

```
                        return false;
                    }
                    // calcola la distanza della cella rispetto all'asse delle x e
                        delle y
                    int deltaX = destinazione.getCoordinata().getX() -
                        this.getCasella().getCoordinata().getX();
                    int deltaY = destinazione.getCoordinata().getY() -
                        this.getCasella().getCoordinata().getY();
                    // se il valore assoluto delle distanze \'e diverso ritorna false
                    if(Math.abs(deltaX)!=Math.abs(deltaY)){
                        return false;
                    }
                    int x = this.getCasella().getCoordinata().getX();
                    int y = this.getCasella().getCoordinata().getY();
                    // controlla tutte le caselle sulla diagonale, se in una di queste
                        c'e' un pezzo ritorna false
                    for(int i=1;i<Math.abs(deltaX);i++){
                        if(!scacchiera.getCasella(
                                new Coordinata(x+Integer.signum(deltaX)*i,
                                    y+Integer.signum(deltaY)*i)).isEmpty()){
                                return false;
                        }
                    }
                    // ritorna true
                    return true;
                }
                return false;
        }

        @Override
        public String toString() {
                return "A ["+this.getColor()+"]";
        }
}
```

Torre

```
package scacchi.scacchiera.pezzi;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Coordinata;
import scacchi.scacchiera.Scacchiera;

public class Torre extends Pezzo {

        public Torre(Casella casella, Color color) {
                super(casella, color);
        }

        @Override
        public boolean mossaValida(Scacchiera scacchiera, Casella destinazione) {
                if (destinazione != null) {
                        // se la casella destinazione non \'e vuota e \'e presente un Pezzo
```

```
                    dello stesso
                // colore ritorna falso
                if (!destinazione.isEmpty()
                            && destinazione.getPezzo().getColor() ==
                                this.getColor()) {
                      return false;
                }
                // se la riga o la colonna non e' costante ritorno false
                int deltaX = destinazione.getCoordinata().getX() -
                    this.getCasella().getCoordinata().getX();
                int deltaY = destinazione.getCoordinata().getY() -
                    this.getCasella().getCoordinata().getY();
                if(!((deltaX==0)||(deltaY==0))){
                      return false;
                }
                if(deltaY==0){
                      // controlla tutte le caselle sulla diagonale, se in una di
                          queste c'e' un pezzo ritorna false
                      for(int i=1;i<Math.abs(deltaX);i++){
                            if(!scacchiera.getCasella(
                                  new Coordinata(
                                      this.getCasella().getCoordinata().getX() + i,
                                      this.getCasella().getCoordinata().getY()
                                      )).isEmpty()){
                                  return false;
                            }
                      }
                }
                if(deltaX==0){
                      for(int i=1;i<Math.abs(deltaX);i++){
                            if(!scacchiera.getCasella(
                                  new
                                      Coordinata(this.getCasella().getCoordinata().getX(),
                                      this.getCasella().getCoordinata().getY()+i)
                                      ).isEmpty()){
                                  return false;
                            }
                      }
                }
                return true;
          }
          return false;
    }

    @Override
    public String toString() {
          return "T ["+this.getColor()+"]";
    }
}
```

Cavallo

```
package scacchi.scacchiera.pezzi;
```

```
import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Scacchiera;

public class Cavallo extends Pezzo {

	public Cavallo(Casella casella, Color color) {
		super(casella, color);
	}

	@Override
	public boolean mossaValida(Scacchiera scacchiera, Casella destinazione) {
		if (destinazione != null) {
			// se la casella destinazione non \'e vuota e \'e presente un Pezzo
			// dello stesso
			// colore ritorna falso
			if (!destinazione.isEmpty()
					&& destinazione.getPezzo().getColor() ==
					    this.getColor()) {
				return false;
			}
			// calcola la distanza della cella rispetto all'asse delle x e delle
			// y
			int deltaX = destinazione.getCoordinata().getX() -
			    this.getCasella().getCoordinata().getX();
			int deltaY = destinazione.getCoordinata().getY() -
			    this.getCasella().getCoordinata().getY();
			if((Math.abs(deltaX)==2 && Math.abs(deltaY)==1) ||
			    (Math.abs(deltaY)==2 && Math.abs(deltaX)==1)){
				return true;
			}
		}
		return false;
	}

	@Override
	public String toString() {
		return "C ["+this.getColor()+"]";
	}
}
```

Donna

```
package scacchi.scacchiera.pezzi;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Coordinata;
import scacchi.scacchiera.Scacchiera;

public class Donna extends Pezzo {

	public Donna(Casella casella, Color color) {
		super(casella, color);
```

```
}

@Override
public boolean mossaValida(Scacchiera scacchiera, Casella destinazione) {
      if (destinazione != null) {
            // se la casella destinazione non \'e vuota e \'e presente un Pezzo
            // dello stesso
            // colore ritorna falso
            if (!destinazione.isEmpty()
                        && destinazione.getPezzo().getColor() ==
                            this.getColor()) {
                  return false;
            }
            // calcola la distanza della cella rispetto all'asse delle x e delle
            // y
            int deltaX = destinazione.getCoordinata().getX() -
                this.getCasella().getCoordinata().getX();
            int deltaY = destinazione.getCoordinata().getY() -
                this.getCasella().getCoordinata().getY();
            // se il valore assoluto delle distanze \'e diverso ritorna false
            if (Math.abs(deltaX) == Math.abs(deltaY)) {
                  // controlla tutte le caselle sulla diagonale, se in una di
                  // queste c'e' un pezzo ritorna false
                  for (int i = 1; i < Math.abs(deltaX); i++) {
                        if (!scacchiera.getCasella(
                                    new
                                        Coordinata(this.getCasella().getCoordinata().getX()
                                                + Integer.signum(deltaX) * i,
                                                    this
                                                .getCasella().getCoordinata().getY()
                                                + Integer.signum(deltaY) *
                                                    i)).isEmpty()) {
                              return false;
                        }
                  }
                  return true;
            }
            int x = this.getCasella().getCoordinata().getX();
            int y = this.getCasella().getCoordinata().getY();
            if ((deltaX == 0) || (deltaY == 0)) {

                  if (deltaY == 0) {
                        // controlla tutte le caselle sulla diagonale, se in
                            una di
                        // queste c'e' un pezzo ritorna false
                        for (int i = 1; i < Math.abs(deltaX); i++) {

                              if (!scacchiera.getCasella(
                                          new Coordinata(x + i, y)).isEmpty()) {
                                    return false;
                              }
                        }
                  }
```

```
                        if (deltaX == 0) {
                              for (int i = 1; i < Math.abs(deltaX); i++) {
                                    if (!scacchiera.getCasella(
                                                new Coordinata(x, y + i)).isEmpty()) {
                                          return false;
                                    }
                              }
                        }
                        return true;
                  }

                  // ritorna true
                  return false;
            }
            return false;
      }

      @Override
      public String toString() {
            return "D ["+this.getColor()+"]";
      }
}
```

Pedone

```
package scacchi.scacchiera.pezzi;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Coordinata;
import scacchi.scacchiera.Scacchiera;

public class Pedone extends Pezzo {

      public Pedone(Casella casella, Color color) {
            super(casella, color);
      }

      @Override
      public boolean mossaValida(Scacchiera scacchiera, Casella destinazione) {
            if (destinazione != null) {
                  // se la casella destinazione non \'e vuota e \'e presente un Pezzo
                  // dello stesso
                  // colore ritorna falso
                  if (!destinazione.isEmpty()
                              && destinazione.getPezzo().getColor() ==
                                  this.getColor()) {
                        return false;
                  }
                  // calcola la distanza della cella rispetto all'asse delle x e delle
                  // y
                  int deltaX = destinazione.getCoordinata().getX() -
                      this.getCasella().getCoordinata().getX();
                  int deltaY = destinazione.getCoordinata().getY() -
```

```
                    this.getCasella().getCoordinata().getY();
                if(this.getColor().equals(Color.BLACK)&&deltaY>0){
                      return false;
                }
                if(this.getColor().equals(Color.WHITE)&&deltaY<0){
                      return false;
                }
                if(Math.abs(deltaY)>1||Math.abs(deltaX)>1){
                      return false;
                }
                if(Math.abs(deltaY)==1&&Math.abs(deltaX)==1&&scacchiera.getCasella(new
                    Coordinata(this.getCasella().getCoordinata().getX()+deltaX,
                    this.getCasella().getCoordinata().getY()+deltaY)).isEmpty()){
                      return false;
                }
                return true;
          }
          return false;
    }

    @Override
    public String toString() {
          return "P ["+this.getColor()+"]";
    }
}
```

2.1.3 Scacchiera

Esercizio 5: Una scacchiera contiene dei pezzi, quando viene creata i pezzi sono posti sulla scacchiera in maniera corretta.

```
package scacchi.scacchiera;

import scacchi.scacchiera.pezzi.Alfiere;
import scacchi.scacchiera.pezzi.Cavallo;
import scacchi.scacchiera.pezzi.Donna;
import scacchi.scacchiera.pezzi.Pedone;
import scacchi.scacchiera.pezzi.Pezzo;
import scacchi.scacchiera.pezzi.Re;
import scacchi.scacchiera.pezzi.Torre;

public class Scacchiera {

    /**
     * 8x8 Matrix of caselle
     */
    private final Casella[][] caselle;
    private static final int SIZE = 8;

    /**
```

```
 * Constructor for Scacchiera, initializes all the fields.
 */
public Scacchiera() {
	// initialize fields

	caselle = new Casella[SIZE][SIZE];
	for (int i = 0; i < SIZE; i++) {
		for (int j = 0; j < SIZE; j++) {
			caselle[i][j] = new Casella(new Coordinata(i, j));
		}
	}

	this.intializeReRow(0, Color.WHITE);
	this.intializePedoniRow(1, Color.WHITE);
	this.intializePedoniRow(6, Color.BLACK);
	this.intializeReRow(7, Color.BLACK);

}

private void intializePedoniRow(int row, Color color) {
	for (int i = 0; i < SIZE; i++) {
		Pezzo pedone = new Pedone(caselle[row][i], color);
		caselle[row][i].setPezzo(pedone);
	}
}

private void intializeReRow(int row, Color color) {
	Pezzo torreBianca1 = new Torre(caselle[row][0], color);
	caselle[row][0].setPezzo(torreBianca1);
	Pezzo cavalloBianca1 = new Cavallo(caselle[row][1], color);
	caselle[row][1].setPezzo(cavalloBianca1);
	Pezzo alfiereBianca1 = new Alfiere(caselle[row][2], color);
	caselle[row][2].setPezzo(alfiereBianca1);
	Pezzo reBianco = new Re(caselle[row][3], color);
	caselle[row][3].setPezzo(reBianco);
	Pezzo reginaBianco = new Donna(caselle[row][4], color);
	caselle[row][4].setPezzo(reginaBianco);
	Pezzo alfiereBianca2 = new Alfiere(caselle[row][5], color);
	caselle[row][5].setPezzo(alfiereBianca2);
	Pezzo cavalloBianco2 = new Cavallo(caselle[row][6], color);
	caselle[row][6].setPezzo(cavalloBianco2);
	Pezzo torreBianca2 = new Torre(caselle[row][7], color);
	caselle[row][7].setPezzo(torreBianca2);
}

public Casella getCasella(Coordinata coordinata) {
	return this.caselle[coordinata.getX()][coordinata.getY()];
}

/**
 * @see java.lang.Object#toString()
 */
@Override
```

```
	public String toString() {
		String ret = "";
		ret += "-----------------------------------------------------";
		ret += "-----------------------------------------------------\n";
		for (int i = 0; i < SIZE; i++) {
			ret += "| ";
			for (int j = 0; j < SIZE; j++) {
				ret += caselle[i][j].toString();
				ret += " | ";
			}
			ret += "\n";
			ret += "-----------------------------------------------------\n";
			ret += "-----------------------------------------------------";
		}
		ret += "\n ";
		return ret;
	}
}
```

2.1.4 Player

Esercizio 6: Nel gioco degli scacchi ci sono due giocatori a ciascuno dei quali è associato un colore

```
package scacchi;

import scacchi.scacchiera.Color;

public class Player {

	private final Color color;

	public Player(Color color){
		this.color=color;
	}

	@Override
	public String toString() {
		return "Player [color=" + color + "]";
	}

	public Color getColor() {
		return color;
	}
}
```

2.1.5 Gioco

Esercizio 1.6 Il gioco consiste in una serie di mosse iterativamente effettuate dai due giocatori

```
package scacchi;

import java.io.IOException;
import java.util.Scanner;

import scacchi.scacchiera.Casella;
import scacchi.scacchiera.Color;
import scacchi.scacchiera.Coordinata;
import scacchi.scacchiera.Scacchiera;
import scacchi.scacchiera.pezzi.Pezzo;

public class Gioco {

	private static final int N_PLAYERS = 2;
	private final Player[] players;
	private final Scacchiera scacchiera;

	public Gioco() {
		players = new Player[N_PLAYERS];
		players[0] = new Player(Color.WHITE);
		players[1] = new Player(Color.BLACK);
		scacchiera = new Scacchiera();
	}

	public void gioca() throws IOException {
		int i = 0;
		while (true) {
			System.out.println("Tocca a " + players[i % 2]);
			Player currPlayer = players[i % 2];
			boolean valid = false;
			Scanner in = new Scanner(System.in);
			do {
				System.out.println(this.scacchiera);
				System.out
						.println("Specifica la riga dove si trova la
                                pedina che vuoi muovere");
				int riga = in.nextInt();
				System.out
						.println("Specifica la colonna dove si trova la
                                pedina che vuoi muovere");
				int colonna = in.nextInt();

				Casella casellaSelezionata = scacchiera
						.getCasella(new Coordinata(riga, colonna));
				if (casellaSelezionata.getPezzo() != null
						&& casellaSelezionata.getPezzo().getColor()
                                .equals(currPlayer.getColor())) {
					Pezzo pezzo = casellaSelezionata.getPezzo();
```

```
                                    System.out
                                                .println("Specifica la riga dove vuoi
                                                    muovere il pezzo");
                                    int rigaDestinazione = in.nextInt();
                                    System.out
                                                .println("Specifica la colonna dove vuoi
                                                    muovere il pezzo");
                                    int colonnaDestinazione = in.nextInt();
                                    Casella
                                        casellaDestinazione=this.scacchiera.getCasella(new
                                        Coordinata(rigaDestinazione, colonnaDestinazione));
                                    if(pezzo.mossaValida(this.scacchiera,
                                        casellaDestinazione)){
                                          casellaSelezionata.unsetPezzo();
                                          casellaDestinazione.setPezzo(pezzo);
                                          pezzo.setCasella(casellaDestinazione);
                                          valid=true;

                                    }
                                    else{
                                          System.out
                                          .println("Non e' possibile muovere la pedina
                                              nella colonna specificata");

                                    }

                              } else {
                                    System.out.println("Nella casella [" + riga + "]["
                                                + colonna
                                                + "] non e' contenuta una pedina del tuo
                                                    colore");
                              }

                        }while(!valid);
                  i++;
                  }
            }
}
```

2.1.6 Main

```
package scacchi;

import java.io.IOException;

public class Main {

      public static void main(String[] args) throws IOException {
            Gioco gioco=new Gioco();
            gioco.gioca();
```

```
        }
}
```

2.2 Persone e Studenti

Esercizio 7: Implementare le classi per rappresentare delle persone e degli studenti.

- Vogliamo tenere traccia del nome e della data di nascita di una persona.
- Gli studenti sono delle persone cui è associata una lista di esami sostenuti.
- Vogliamo controllare se lo studente ha abbastanza crediti per laurearsi.
- Vogliamo fornire uno strumento per il calcolo della media pesata

2.2.1 Raffinamento

Il primo passo da effettuare consiste nell'identificare le entità in gioco. In questo caso, supponiamo vengano considerate le entità di persona, studente e voto. Per ognuna di queste si dovrà implementare la relativa classe che ne modelli il comportamento. Avremo quindi le classi `Person`, `Student` e `Grade`.

2.2.2 Persona

- quali funzionalità deve fornire una `Persona`?
  - `String getName()`: ritorna il nome della persona
  - `Date getBirthday()`: ritorna il compleanno della pesona
- come memorizzo lo stato degli oggetti di tipo persona?
  - `String name`: contiene il nome della persona
  - `Date birthday`: contiene la data di compleanno di una persona
- come è strutturato il costruttore di un oggetto persona?
  - `Person(String name, Date birthday)`: crea una persona con un determinato nome e una data data di compleanno
- Ha senso che un elemento di tipo persona sia `immutabile`?

```
import java.util.Date;

public class Person {
      private final String name;
      private final Date birthday;
      public Person (String name, Date birthday){
            this.name = name;
            // creo una copia dell'oggetto date per evitare che venga modificato
                dall'esterno
            this.birthday = new Date(birthday.getTime());
```

```
	}

	public String getName(){
		return name;
	}
	// anche qui creo una copia per evitare modifiche esterne
	public Date getBirthday(){
		return new Date(birthday.getTime());
	}
	@Override
	public String toString(){
		return "My name is " + name;
	}
}
```

2.2.3 Grade

- quali funzionalità deve fornire un `Grade`?
  - `String getSubject()`: ritorna la materia corrispondente al grade
  - `int getPoints()`: ritorna il punteggio ottenuto
  - `int getCredits()`: ritorna i crediti dell'esame
- come memorizzo lo stato degli oggetti di tipo persona?
  - `String subject`: contiene il nome dell'esame
  - `int points`: contiene il punteggio ottenuto
  - `int credits`: contiene i crediti relativi all'esame
- come è strutturato il costruttore di un oggetto Grade?
  - `Grade(String subject, int points, int credits)`: crea un Grade che si riferisce a una determinata materia, con un determinato punteggio e un dato numero di crediti
- Ha senso che un elemento di tipo Grade sia `immutabile`?

```
import java.util.Date;

public class Grade {
	private final String subject;  // La materia dell'esame
	private final int points;     // La valutazione ricevuta
	private final int credits;    // Il valore in crediti dell'esame

	public Grade(String subject, int points, int credits) {
	   // che cosa succede se il punteggio e' minore di 18 e se e' maggiore di 30?
		this.subject = subject;
		this.points = points;
		this.credits = credits;
	}

	public String getSubject() {
		return subject;
```

```
	}

	public int getPoints() {
		return points;
	}

	public int getCredits() {
		return credits;
	}
}
```

2.2.4 Studente

- quali funzionalità deve fornire uno `Studente`?
  - `void addGrade(Grade grade)`: aggiunge una valutazione alla carriera dello studente
  - `boolean canGraduate()`: controlla se lo studente può graduarsi
  - `double getWeightedGradeAverage()`: ritorna la media pesata dello studente
- come memorizzo lo stato degli oggetti di tipo Studenti? Nota è ragionevole che lo studente estenda Persona
  - `List<Grade>`: contiene la lista degli esami effettuati dallo studente
- come è strutturato il costruttore di un oggetto Studente?
  - `Student (String name, Date birthday)`: crea uno studente con un dato nome e data di nascita
- Ha senso che un elemento di tipo Studente sia `immutabile?`: No, gli esami associati allo studente devono cambiare (anche se uno sviluppatore potrebbe comunque decidere di implementarlo mediante una classe immutabile.

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Student extends Person{
	private final List<Grade> grades;

	public Student (String name, Date birthday){
		super(name,birthday);
		grades = new ArrayList<Grade>();
	}

	// Aggiunge una valutazione alla carriera dello studente.
	public void addGrade(Grade grade) {
		grades.add(grade);
	}

	// Controlla se lo studente ha abbastanza crediti per potersi laureare.
	public boolean canGraduate() {
		return totalCredits() >= 180;
	}
```

```
	// Calcola la media pesata.
	public double getWeightedGradeAverage() {
		double sumOfWeightedPoints = 0;
		for (Grade grade : grades) {
			sumOfWeightedPoints += grade.getCredits() * grade.getPoints();
		}
		return sumOfWeightedPoints / totalCredits();
	}

	// Calcola il numero di crediti sostenuti dallo studente.
	private int totalCredits() {
		int totalCredits = 0;
		for (Grade grade : grades) {
			totalCredits += grade.getCredits();
		}
		return totalCredits;
	}

	@Override
	public String toString() {
		return super.toString() + " and I am a student";
	}
}
```

2.2.5 Main Esempio

```
import java.util.Date;

/**
 * classe di esempio per mostrare
 * la differenza tra tipi statici e dinamici
 * @author valerio
 *
 */
public class Esempio {
	public static void main(String[] args){
		// Il costruttore di Date e' deprecato, lo usiamo
		// solo per comodita' in questo esempio.
		Person bill = new Person("Bill", new Date(1955,10,28));
		Person pippo = new Student("Pippo", new Date(1930,05,02));
		System.out.println(bill);
		System.out.println(pippo);
	}
}
```

- Qual è il po statico di ciascuna variabile?: *il tipo statico di bill e di pippo è Person*
- Qual è il tipo dinamico (dopo l'inizializzazione)? *Il tipo dinamico di bill è persona, mentre di pippo è studente*
- Cosa stampa? Perchè? *Stampa "My name is Bill" e "My name is Pippo and I am a student"*

- Cosa succederebbe se scrivessi l'istruzione pippo.canGraduate() *Errore di compilazione, il tipo statico di Pippo è persona, quindi il compilatore cerca nel tipo* `Person` *la signature del metodo* `canGraduate` *che non è presente.*

2.3 Forme Geometriche

Esercizio 8: Implementare una gerarchia di classe che rappresenti delle forme geometriche, permettendo di calcolare

- La loro area
- Perimetro
- E di rappresentarle sullo schermo.

2.3.1 TwoDimensionalCanvas

Perchè usiamo un interfaccia per modellare TwoDimensionalCanvas?
Una classe (anche astratta) ci dice che un oggetto è qualcosa, una interfaccia rappresenta un comportamento che la classe ha (indipendentemente da come è fatto l'oggetto). Non conosciamo nulla su come è fatto TwoDimensionalCanvas sappiamo solamente che alcuni oggetti avranno le funzionalità specificate in TwoDimensionalCanvas

```
public interface TwoDimensionalCanvas {
	void drawPoint(Point point);

	void drawLine(Point firstEndPoint, Point secondEndPoint);

	void drawCircle(Point center, double radius);
}
```

2.3.2 Point

```
public class Point {
	private final double x;
	private final double y;

	public Point(double x, double y) {
		this.x = x;
		this.y = y;
	}

	public double getX() {
		return x;
	}

	public double getY() {
		return y;
	}

	public Point rotate(Point center, double degrees) {
```

```
            double rotatedX = center.x + (x - center.x) * Math.cos(degrees) - (y -
                center.y) * Math.sin(degrees);
            double rotatedY = center.y + (x - center.x) * Math.sin(degrees) + (y -
                center.y) * Math.cos(degrees);
            return new Point(rotatedX, rotatedY);
      }

      public String toString(){
            return "x = " + x + " y = " + y;
      }
}
```

2.3.3 Shape

Prima di tutto definiamo la classe astratta `Shape`. Perchè usiamo una classe astratta?
Una classe (anche astratta) ci dice che un oggetto è qualcosa, una interfaccia rappresenta un comportamento che la classe ha (indipendentemente da come è fatto l'oggetto)

- quali funzionalità deve fornire uno `Shape`?
  - `double getArea()`: ritorna l'area della figura
  - `double getPerimeter()`: ritorna il perimetro della figura
  - `List<Point> getSequenceOfPointsToDraw();`: ritorna i punti da disegnare
  - `public void draw(TwoDimensionalCanvas canvas)`: stampa la figura su un particolare canvas

```
import java.util.List;

public abstract class Shape {
      public abstract double getArea();
      public abstract double getPerimeter();
      public abstract List<Point> getSequenceOfPointsToDraw();

      public void draw(TwoDimensionalCanvas canvas){
            List<Point> points = getSequenceOfPointsToDraw();
            for (int i = 0; i < points.size() - 1; i++ ){
                  canvas.drawLine(points.get(i), points.get(i + 1));
            }
      }
}
```

2.3.4 Circle

```
import java.util.ArrayList;
import java.util.List;

public class Circle extends Shape{
      private final Point center;
      private final double radius;
```

```
	public Circle(Point center, double radius) {
		his.center = center;
		this.radius = radius;
	}

	@Override
	public double getArea() {
		return Math.PI * radius * radius;
	}

	@Override
	public double getPerimeter() {
		return 2 * Math.PI * radius;
	}

	@Override
	public List<Point> getSequenceOfPointsToDraw() {
		// Primo approccio:
		// Approssimiamo il cerchio con una figura regolare con 256 lati
		List<Point> points = new ArrayList<Point>();
		int numberOfSegments = 256;
		for (int i = 0; i < numberOfSegments; i++) {
			double newPointX = center.getX() + radius * Math.cos(2 * Math.PI *
			    i / numberOfSegments);
			double newPointY = center.getY() + radius * Math.sin(2 * Math.PI *
			    i / numberOfSegments);
			points.add(new Point(newPointX, newPointY));
		}
		return points;
	}

	@Override
	public void draw(TwoDimensionalCanvas canvas) {
		/* Secondo approccio: overriding di draw
		 * Seguendo questo approccio, possiamo utilizzare le primitive per
		 * disegnare messe a disposizione dal piano. L'introduzione di una nuova
		 * forma richiede un minor numero di modifiche al codice.
		 */
		canvas.drawCircle(center, radius);
	}
}
```

2.3.5 Square

```
import java.util.ArrayList;
import java.util.List;

public class Square extends Shape{

	private final Point initialPoint;
```

```
	// It represents the orientation, in radiant degrees of the square
	private final double orientation;
	private final double sideLength;

	public Square(Point initialPoint, double orientation, double sideLength) {

		this.initialPoint = initialPoint;
		this.orientation = orientation;
		this.sideLength = sideLength;
	}

	@Override
	public double getArea() {
		return sideLength * sideLength;
	}

	@Override
	public double getPerimeter() {
		return 4 * sideLength;
	}

	@Override
	public List<Point> getSequenceOfPointsToDraw() {
		List<Point> points = new ArrayList<Point>();
		points.add(initialPoint);
		Point nonOrientedSecondPoint = new Point(initialPoint.getX(),
		    initialPoint.getY() + sideLength);
		points.add(nonOrientedSecondPoint.rotate(initialPoint, orientation));
		Point nonOrientedThirdPoint = new Point(initialPoint.getX() + sideLength,
		    initialPoint.getY() + sideLength);
		points.add(nonOrientedThirdPoint.rotate(initialPoint, orientation));
		Point nonOrientedFourthPoint = new Point(initialPoint.getX() +
		    sideLength, initialPoint.getY());
		points.add(nonOrientedFourthPoint.rotate(initialPoint, orientation));
		return points;
	}
}
```

2.3.6 TextualTwoDimensionalCanvas

```
public class TextualTwoDimensionalCanvas implements TwoDimensionalCanvas{

	@Override
	public void drawPoint(Point point) {
		System.out.println("drawing point: " + point);

	}

	@Override
	public void drawLine(Point firstEndPoint, Point secondEndPoint) {
		System.out.println("drawing Line from: " +
	firstEndPoint + " to: " + secondEndPoint);
```

```
	}

	@Override
	public void drawCircle(Point center, double radius) {
		System.out.println("drawing circle with center: " +
	center + " and radius: " + radius);

	}

}
```

2.3.7 ShapeClient

```
public class ShapeClient {
	public static void main(String[] args){
		Shape s1 = new Square(new Point(0,0), 0 ,10);
		Shape s2 = new Circle(new Point(0,0),5);
		TwoDimensionalCanvas canvas = new TextualTwoDimensionalCanvas();
		s1.draw(canvas);
		s2.draw(canvas);
	}
}
```

2.4 Forme Geometriche (Resizable, Rotatable)

Esercizio 9: Deve essere possibile ruotare e ridimensionare gli oggetti shape. A seconda della loro natura (e.g., un cerchio non può essere ruotato ma può essere ridimensionato)

2.4.1 Rotatable

```
public interface Rotatable {
	public Shape rotate(double angle);
}
```

2.4.2 Resizable

```
public interface Resizable {
	public Shape resize(double scale);
}
```

2.4.3 Square

```
public class Square extends Shape implements Resizable, Rotatable{
```

```
    //...
     @Override
      public Square rotate(double angle) {
           return new Square(initialPoint, orientation+angle, sideLength);
      }
      @Override
       public Square resize(double scale) {
           return new Square(initialPoint, orientation, sideLength*scale);
      }
}
```

2.4.4 Circle

```
public class Circle extends Shape implements Resizable {
   //..
   @Override
   public Circle resize(double scale) {
   return new Circle(center, radius*scale);
   }
}
```

3 Curiosità

In un paper intitolato "Java: an Overview", James Gosling nel 1995 discute le ragioni per cui l'ereditarietà multipla non è supporta in Java. *JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions.*

4 Esercizi per casa

- Esercizio 1:
  - Modificare la soluzione dell'esercizio relativo agli scacchi al fine di:
    - *rimuovere le duplicazioni di codice*. In generale è sempre bene ridurre al minimo le parti di codice duplicato. All'interno del metodo `mossavalida(..)` i vari pezzi controllano che la casella destinazione non è uguale a `null`, che non è presente un pezzo dello stesso colore etc.. Implementare una variante nel quale le replicazioni di codice sono ridotte..
    - *nell'implementazione corrente non gestisce correttamente la mossa iniziale del pedone, ovvero il fatto che il pedone inizialmente può muoversi di due caselle. Rimuovere questa limitazione*
    - *implementare la mossa dell'arrocco*
    - *implementare la mossa speciale dell'en passant*
- Esercizio 2:
  - Modificare la soluzione dell'esercizio dell'esercizio persone e studenti al fine di
    - *aggiungere la possibilità di cambiare il voto agli esami*

* *aggiungere un'entità che rappresenta un docente. A ogni grade è associato uno e un solo docente e a ogni docente è associata la lista di grade assegnati*
* *È sempre consigliabile non usare tipi e metodi deprecati. Reimplementare l'esercizio persone e studenti utilizzando la classe* `GregorianCalendar` *invece della classe* `Date`*.*