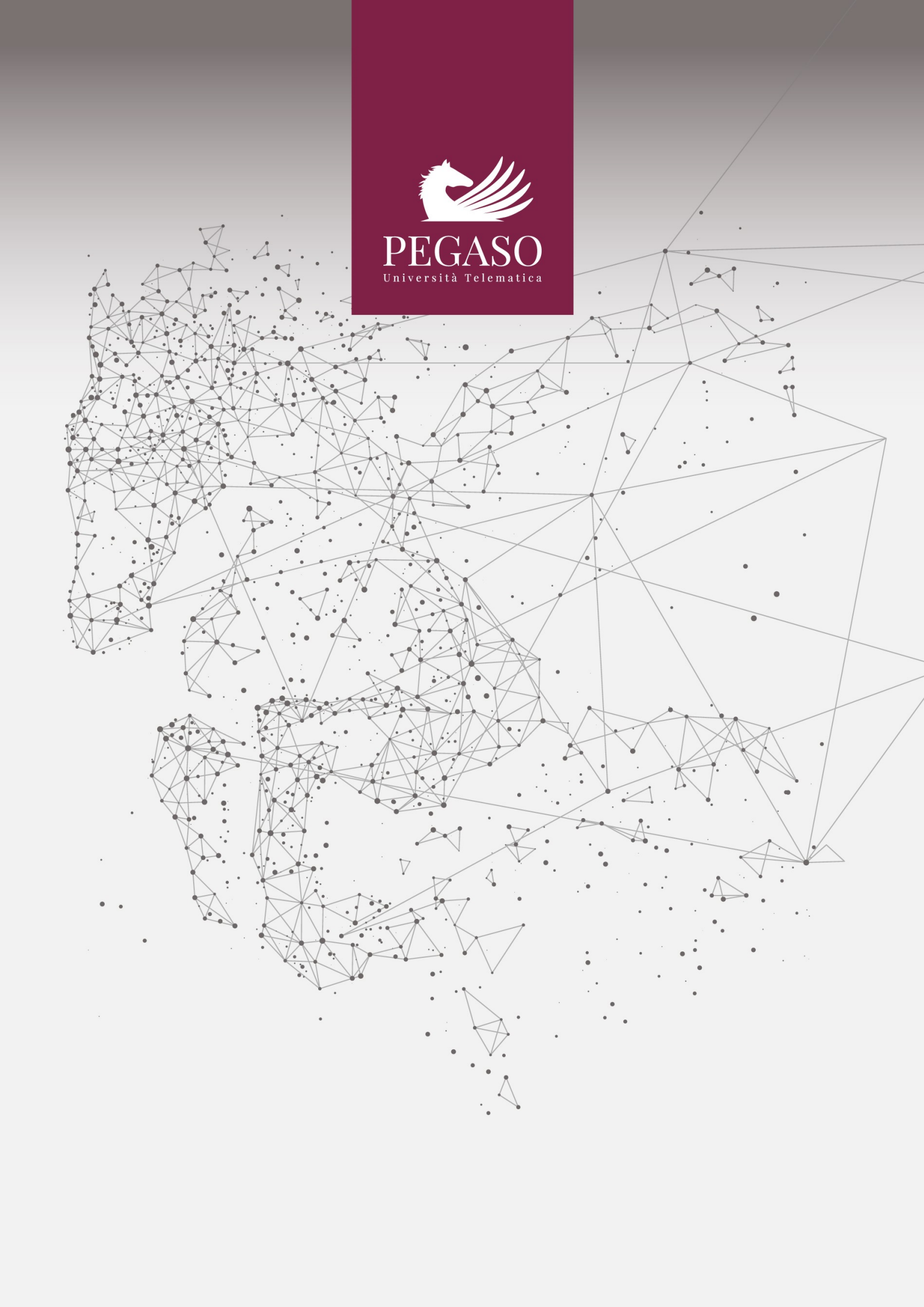
PEGASO
Università Telematica

Indice

1. Polimorfismo

Il polimorfismo (dal greco «molte forme») è un altro concetto che dalla realtà è stato importato nella programmazione ad oggetti.

In generale esso permette di riferirci con un unico termine ad entità diverse.

In Java, il polimorfismo è la capacità di oggetti di classi diverse di rispondere a un metodo in modo diverso, in base alla loro implementazione specifica.

Infatti, il metodo effettivo invocato sarà quello della classe dell'oggetto effettivamente istanziata grazie ad un meccanismo binding dinamico (oppure il late binding).

Grazie al late binding il metodo appropriato da invocare viene determinato solo in fase di esecuzione quindi dinamicamente.

Si parla infatti di polimorfismo dinamico che si riferisce alla scelta del metodo da invocare eseguita in fase di esecuzione sulla base del tipo effettivo dell'oggetto istanziato, e non sulla base del tipo di riferimento all'oggetto.

Il polimorfismo dinamico si verifica in generale quando si utilizza l'ereditarietà e l'override dei metodi.

Infatti, il polimorfismo è reso possibile dal fatto che una classe figlia può essere trattata come una classe padre.

Questo significa che un oggetto di una classe figlia può essere assegnato a una variabile di tipo della classe padre, consentendo al codice di chiamare i metodi della classe figlia attraverso la variabile di tipo della classe padre. Inoltre, il polimorfismo si attua anche tramite l'uso di metodi astratti, classi astratte e interfacce.

2. Classi astratte

Analizziamo questo componente come importante strumento di progettazione e di realizzazione del polimorfismo dinamico.

Metodo astratto

Un metodo astratto in Java è solo dichiarato all'interno della classe e non implementa un proprio blocco di codice.

Un metodo astratto fornisce solo la firma (signature) di metodo, che specifica il nome, il tipo di ritorno, i parametri, ma non fornisce alcuna implementazione del metodo stesso.

L'implementazione deve essere fornita dalle sottoclassi che estendono la classe contenente il metodo astratto.

```
...
public abstract void dipingiQuadro();
...
```

Fig. 1: *Dichiarazione di un metodo astratto*

Come mostrato in Fig. 1, un metodo astratto viene dichiarato utilizzando la parola chiave "abstract" prima della firma del metodo, senza fornire un'implementazione del corpo del metodo.

In questo esempio, si dichiara un metodo astratto dipingiQuadro() che restituisce un void, ma non viene fornita alcuna implementazione per il metodo.

Questo significa che ogni sottoclasse che estende la classe in cui questo metodo è dichiarato deve fornire un'implementazione concreta del metodo dipingiQuadro().

Classe astratta

Una classe astratta è una classe che non può essere istanziata direttamente, ma può essere utilizzata solo come superclasse per altre classi.

Una classe astratta può contenere metodi astratti, ovvero metodi che non forniscono un'implementazione concreta, ma solo una firma di metodo che deve essere implementata dalle sottoclassi.

Inoltre, una classe astratta può contenere metodi concreti (cioè implementati), attributi, costruttori e altre componenti delle classi.

Se non si possono istanziare perché creare una classe astratta?

Il seguente esempio mostrato di utilizzo di una classe astratta chiarirà l'utilità delle classi astratte (vedi Fig. 2).

L'esempio considera una classe Pittore con un metodo definito astratto che è DipingiQuadro() e quindi la classe stessa è definita come astratta.

Per fornire una implementazione di DipingiQuadro() dovremmo saper rispondere alla domanda: «Con quale stile disegnare?».

La risposta più sensata a questa domanda sarebbe «Dipende dal pittore.

Se il pittore è un impressionista avrà un certo stile nel disegnare, se è invece un neo-realista ne avrà un altro».

Questo «dipende da…» corrisponde a diverse implementazioni possibili del metodo in questione in classi specializzate derivate dalla classe Pittore.

Invece di fornire una definizione «vuota» o «inventata» di un metodo che si pensa di ridefinire in una sottoclasse si può dichiarare il metodo astratto come segue:

public abstract void dipingiQuadro(); e poi si fornisce l'implementazione specifica nelle classi specializzate PittoreImpressionista e PittoreNeoRealista.

```
public abstract class Pittore {
        //variabili di istanza
        //costruttore
        //metodi
        public abstract void dipingiQuadro();
        //altri metodi…
}

public class PittoreImpressionista extend Pittore {
//con implementazione specifica del metodo
//dipingiQuadro
}

public class PittoreNeoRealista extend Pittore {
//con implementazione specifica del metodo
//dipingiQuadro
}
```

Fig. 2: *Esempio di utilizzo di una classe astratta*

Definire un metodo astratto significa posticipare la sua implementazione al momento in cui si saprà effettivamente come farlo.

Nel caso specifico è come dire che «ogni pittore avrà un suo stile in base alla corrente pittorica di riferimento e di preferenza, adesso in questa classe o in questa fase di costruzione del software non si sa qual è e quindi si lascia non implementato».

D'altro canto in una prima fase di progettazione può bastare ragionare in termini della classe generica Pittore e poi pensare ai dettagli in una fase successiva del progetto quando le classi devono essere definite perché siano implementate da uno sviluppatore.

Quindi in un primo momento non ha senso o non c'è proprio il bisogno di istanziare una classe Pittore.

In un momento successivo poi si popolerà il sistema di sottoclassi Pittore più specifiche come PittoreImpressionista o PittoreNeoRealista che implementano il metodo astratto dipingiQuadro().

Si noti che la classe Pittore avrebbe potuto implementare concretamente il metodo dipingiQuadro(), con una implementazione di default.

Tuttavia a livello logico non sarebbe stato corretto favorire l'implementazione di uno stile pittorico piuttosto che un altro.

Sebbene la classe Pittore sia dichiarata astratta per via del metodo astratto dipingiQuadro(), altri metodi della classe astratta possono essere tuttavia definiti ed è bene farlo tutte le volte che è possibile ed ha un senso.

In altre parole, quando esiste una specifica per la definizione di un metodo, questo va implementato.

In tal modo la classe base conterrà tutti quei dettagli già noti e che non dovranno poi essere ridefiniti nelle classi derivate.

Ricapitolando, le classi astratte sono un importante strumento di progettazione software perché consentono di definire delle classi di base che possono essere utilizzate per creare (tramite ereditarietà) nuove classi più specifiche e specializzate.

In particolare, le classi astratte definiscono delle funzionalità comuni a un insieme di classi correlate, funzionalità che poi le classi derivate (o sottoclassi) devono implementare in modo specifico.

In questo modo, le classi astratte consentono di definire un'interfaccia comune per un gruppo di classi, semplificando così la progettazione e la manutenzione del codice.

Inoltre, l'utilizzo delle classi astratte può ridurre la complessità del codice e aumentare la modularità del software, facilitando la sua estendibilità e il riutilizzo del codice essendo anche utilizzate in combinazione con in concetti di ereditarietà e polimorfismo.

Le classi astratte inducono a pensare per classi generiche o per superclassi e quindi a pensare al software per astrazioni andando anche a forzare la creazione di sottoclassi dove si andranno a realizzare e definire i dettagli.

3. Interfacce

Da un punto di vista della progettazione, un'interfaccia è un'evoluzione del concetto di classe astratta.

Dal punto di vista del codice, somiglia invece ad una classe senza la sua implementazione interna.

In Java infatti, l'interfaccia specifica una collezione di metodi astratti (senza implementazione) che una classe che realizza quella interfaccia deve implementare.

Le classi che implementano un'interfaccia devono quindi fornire un'implementazione concreta per tutti i metodi astratti definiti nell'interfaccia.

In altre parole, un'interfaccia è un contratto che descrive i comportamenti che una classe deve implementare se vuole soddisfare l'interfaccia.

Un'interfaccia Java inizia come una definizione di classe, tranne per il fatto che utilizza la parola riservata interface al posto di class, come mostrato in Fig. 3.

```
// Una interfaccia che definisce un comportamento comune per
oggetti geometrici (cerchi, quadrati, triangoli)
public interface Forma {
        public double getPerimetro(); // Restituisce il perimetro.
        public double getArea{); //Restituisce l'area
        //altri metodi …
}

//Una interfaccia che offre un comportamento comune per oggetti
che eseguono conversioni fra unità di misura differenti
public interface Convertitore {
        public static final int POLLICI_PER_PIEDE = 12;
        public static doublé convertiInPollici(double piedi);
        public static doublé convertiInPiedi(double pollici);
        //altri metodi …
}
```

Fig. 3: *Esempi di interfacce*

In Fig. 3 si può vedere l'esempio di una interfaccia denominata Forma che definisce un comportamento comune per oggetti geometrici come cerchi, rettangoli, triangoli e così via.

Essa contiene metodi per calcolare l'area e il perimetro della forma ma potrebbe avere ulteriori metodi ad esempio per disegnare la forma su uno schermo o su una stampante.

Per convenzione, il nome di un'interfaccia inizia con una lettera maiuscola, proprio come il nome dì una classe.

L'interfaccia può contenere un numero qualsiasi di intestazioni di metodi pubblici, ognuna seguita da un punto e virgola, e può contenere poi ovviamente commenti.

Un'interfaccia non dichiara alcun costruttore.

I metodi di un'interfaccia devono essere pubblici.

Essa non contiene, tuttavia, variabili di istanza o definizioni complete di metodo; in pratica, i metodi non possono avere un corpo.

Un'interfaccia può anche definire un numero qualsiasi di costanti pubbliche come si può vedere nell'esempio sotto di una interfaccia denominata Convertitore che astrae il concetto ovvero il comportamento di convertibilità tra piedi e pollici.

A partire da Java 8, un'interfaccia può contenere anche metodi statici e di default.

Si può memorizzare un'interfaccia in un file distinto, utilizzando un nome che inizia con il nome dell'interfaccia, seguito da .java. Per esempio, l'interfaccia presentata nel listato di Fig.3 sopra è salvata nel file Forma.java.

In Java, le interfacce sono definite in pratica come classi astratte che contengono solo metodi astratti e costanti.

Le classi che implementano un'interfaccia devono fornire un'implementazione concreta per tutti i metodi astratti definiti nell'interfaccia in modo da creare una classe concreta che può essere istanziata (in caso contrario sarà una classe astratta).

Un esempio è mostrato in Fig. 4 che mostra una classe Quadrato che implementa l'interfaccia Forma.

Le classi che implementano l'interfaccia Forma devono dichiararlo nella intestazione tramite la parola chiave implements e dovranno quindi fornire l'implementazione di questi due metodi.

In questo caso abbiamo considerato la classe Quadrato che implementa il comportamento Forma aderendo quindi ad un certo contratto definito dai metodi getArea() e getPerimetro() che andrà quindi implmentare (oltre al suo costruttore).

Adesso la classe Quadrato è una classe concreta che può essere istanziata.

Perché utilizzare le interfacce?

Anche le interfacce come le classi astratti sono un importante strumento di progettazione e inducono a pensare per astrazioni e non per implementazioni specifiche.

Le interfacce sono utilizzate in Java per creare un alto livello di astrazione, separando l'implementazione dei dettagli dell'interfaccia stessa.

Ciò permette di sviluppare il software in modo modulare e flessibile, in quanto le classi possono implementare molteplici interfacce e fornire implementazioni personalizzate per ogni interfaccia.

In sostanza le interfacce in Java definiscono un contratto tra una classe e l'utente della classe (ovvero chi andrà ad usare la classe).

Scrivere un'interfaccia è un modo con cui il progettista di una classe specifica i metodi a un altro programmatore.

Implementare un'interfaccia è un modo per un programmatore per aderire al contratto ed implementare determinati metodi secondo una certa astrazione pensata per risolvere il problema.

In Java, sia una classe astratta che un'interfaccia possono definire metodi astratti e quindi come tali non possono essere istanziate.

Il vantaggio comune che offrono sia le classi astratte che le interfacce risiede nel fatto che esse possono obbligare le sottoclassi ad implementare comportamenti.

Una classe che eredita un metodo astratto infatti deve effettuare un override del metodo ereditato oppure essere dichiarata a sua volta astratta.

In modo simile per una classe che implementa una interfaccia, essa deve offrire l'implementazione dei metodi astratti dell'interfaccia per essere una classe concreta.

Tuttavia ci sono alcune differenze importanti tra le due:

- Una classe astratta può contenere anche metodi non astratti, mentre un'interfaccia può contenere solo metodi astratti e costanti.
- Una classe può implementare più interfacce, ma può estendere solo una classe astratta.
- Solo una classe può estendere un'altra classe, non gli altri tipi di Java fra cui in particolare l'interfaccia. Le interfacce possono estendere altre interfacce, ma non possono estendere classi (astratte o concrete).
- Una classe astratta può avere costruttori, mentre un'interfaccia non può averli.
- In una classe astratta, i metodi astratti possono essere definiti come public o protetti, mentre in un'interfaccia tutti i metodi sono implicitamente public.
- Le classi astratte sono utilizzate principalmente per definire una classe base, che può essere estesa da altre classi per ereditare dei comportamenti specifici definiti da metodi (troppo generici per essere specificato nel contesto in cui si dichiara), mentre le interfacce sono utilizzate per definire un comportamento più generale espresso attraverso un contratto che deve essere implementato dalle classi che vogliono adottare quel comportamento fornendo implementazioni diverse dei metodi definiti nell'interfaccia. Un'interfaccia è più quindi un'astrazione comportamentale a livell di oggetto.

In generale, se si desidera definire una gerarchia di classi, si utilizza una classe astratta, mentre se si desidera definire un contratto comune tra classi non correlate tra loro, si utilizza un'interfaccia.

Dopo la definizione generale di polimorfismo e dopo aver visto due importanti strumenti di progettazione, vediamo il ruolo che hanno nel polimorfismo e come usarli in un software orientato agli oggetti in combinazione con l'ereditarietà per scrivere codice flessibile e modulare in gradi di adattarsi a diverse situazioni.

Come detto le classi/metodi astratti e le interfacce sono strettamente correlati in Java al polimorfismo rappresentandone un importante strumento.

Le classi astratte possono definire metodi astratti che devono essere implementati dalle sottoclassi.

Lo stesso dicasi per le interfacce che presentano solo metodi astratti rappresentando di fatto l'astrazione di un comportamento.

Questo meccanismo consente di definire dei metodi comuni o un'interfaccia comune per un insieme di classi correlate, ma lascia alle classi derivate il compito di implementare le funzionalità specifiche.

Quando una sottoclasse implementa un metodo astratto ereditato dalla classe astratta, il metodo viene automaticamente reso disponibile attraverso il meccanismo del polimorfismo.

In pratica, il polimorfismo dinamico consente di utilizzare gli oggetti delle sottoclassi come se fossero oggetti della classe (base) astratta o dell'interfaccia che le sottoclassi implementano.

In questo modo, è possibile scrivere codice che lavora sulla classe base o sull'interfaccia comune e che può essere riutilizzato con diverse implementazioni specifiche.

In sostanza, l'uso di classi astratte e interfacce permette di creare codice polimorfico, che può essere utilizzato per scrivere metodi che accettano oggetti di classi diverse ma che implementano la stessa interfaccia, senza però sapere in anticipo il tipo specifico dell'oggetto.

Supponiamo di avere una classe astratta Veicolo che dichiara un metodo astratto calcolaVelocità(). Si hanno poi un attributo velocità e il costruttore.

Abbiamo anche due classi concrete che estendono la classe astratta Veicolo: Automobile e Moto. Entrambe le classi implementano il metodo astratto calcolaVelocità() in modo diverso: per la classe Automobile viene restituito il valore dell'attributo velocità, per la classe Moto viene restituito il valore dato dall'espressione numerica velocità sommato a cilindrata moltiplicato per 0.1.

```
public abstract class Veicolo {
    protected double velocita;
    public Veicolo(double velocita) {
        this.velocita = velocita;
    }
    public abstract double calcolaVelocita(); //metodo astratto
}

public class Automobile extends Veicolo {
    public Automobile(double velocità) {
        super(velocita); //invocazione del costruttore classe base
    }
    public double calcolaVelocità() {
        return velocita;
    } // implementazione del metodo astratto
}

public class Moto extends Veicolo {
    private int cilindrata;
    public Moto(double velocità, int cilindrata) {
        super(velocità); //invocazione del costruttore classe base
        this.cilindrata = cilindrata;
    }
    public double calcolaVelocità() {
        return velocita + (cilindrata * 0.1);
    } // implementazione del metodo astratto
}
```

Fig. 4: *Esempio di polimorfirsmo e classi astratte*

Ora possiamo utilizzare il polimorfismo per creare un array di Veicolo che contiene oggetti di entrambe le sottoclassi, come mostrato in Fig. 5.

```
public class Main {
    public static void main(String[] args) {
        Veicolo[] veicoli = new Veicolo[2]; // array di veicoli
        veicoli[0] = new Automobile(120); // oggetto della classe Automobile
        veicoli[1] = new Moto(80, 600); // oggetto della classe Moto

        for (Veicolo veicolo : veicoli) {
            System.out.println("Velocità: " + veicolo.calcolaVelocità());
        }
        //invocazione del metodo calcolaVelocità della classe Automobile e Moto
    }
}
```

Fig. 5: *Esempio di utilizzo di polimorfirsmo e classi astratte (main)*

In questo modo, quando richiamiamo il metodo calcolaVelocità() su ciascun oggetto dell'array, viene eseguita l'implementazione appropriata del metodo per la sottoclasse di ogni oggetto.

Questo è un esempio di polimorfismo in azione, in cui oggetti di classi diverse rispondono allo stesso messaggio in modi diversi. Si osservi abbiamo usato la versione compatta Veicolo veicolo: veicoli per scorrere l'array.

Per estensione quanto visto può essere applicato anche alle interfacce (vedi Fig. 6).

Supponiamo di usare ancora una interfaccia Forma con i metodi astratti getArea()e getPerimetro(). Abbiamo anche due classi concrete che implementano l'interfaccia Figura: Cerchio e Quadrato.

Entrambe le classi implementano i metodi getArea() e getPerimetro() in modo diverso come mostrato in Fig. 6.

Ora possiamo utilizzare il polimorfismo per creare un array di Figura che contiene oggetti di entrambe le classi, come mostrato in Fig.7.

In questo modo, quando richiamiamo il metodo calcolaArea() su ciascun oggetto dell'array, viene eseguita l'implementazione appropriata del metodo per la classe di ogni oggetto.

Questo è un esempio di polimorfismo in azione, in cui oggetti di classi diverse implementano lo stesso metodo dichiarato nell'interfaccia Figura.

```
// Una interfaccia che definisce un comportamento comune per
oggetti geometrici (cerchi, quadrati, triangoli)

public interface Forma {
        public double getPerimetro(); // Restituisce il perimetro.
         public double getArea(); //Restituisce l'area
         //altri metodi …
}

// Le classi Quadrato e Cerchio che implementano l'interfaccia
// Forma andando a definire i metodi getPerimetro e getArea

public class Quadrato implements Forma {
        private double lato;

        public Quadrato(double ilLato) {
        super();
        lato = ilLato;
        }

        public double getArea() {
        return lato * lato;
        }

        public double getPerimetro() {
        return lato * 4;
        }
}
public class Cerchio implements Forma {
        private double raggio;

        public Quadrato(double ilRaggio) {
        super();
        raggio = ilRaggio;
        }

        public double getArea() {
        return Math.PI * Math.pow(raggio,2);
        }

        public double getPerimetro() {
        return 2*Math.PI*raggio;
        }
}
```

Fig. 6: *Esempio di polimorfismo e interfacce*

```
public class Main {
    public static void main(String[] args) {
        Forma[] figure = new Forma[2];
        figure[0] = new Cerchio(5);
        figure[1] = new Quadrato(10);

        for (Figura figura : figure) {
               System.out.println("Area: " + figura.getArea());
               System.out.println(«Perimetro: " + figura.getPerimetro());

        }
    }
}
```

Fig. 7: *Esempio di utilizzo di polimorfirsmo e interfacce (main)*

Il punto chiave in entrambi gli esempi è il seguente: il polimorfismo dinamico consente di utilizzare gli oggetti delle sottoclassi come se fossero oggetti della classe (base) astratta o dell'interfaccia.

In entrambi gli esempi abbiamo basato il ragionamento sull'esistenza delle classi astratte o interfacce andando a creare un vettore rispettivamente di oggetti di tipo Veicolo o Forma e iterando su essi. In sostanza abbiamo usato gli oggetti delle sottoclassi come fossero oggetti della classe base astratta oppure dell'interfaccia e costruito una logica del software sulla base di essi senza dover sapere in anticipo il tipo specifico dell'oggetto.

Siccome in virtù della new gli oggetti effettivamente istanziati sono Cerchio e Quadrato allora le invocazioni sono fatte sulle loro implementazioni specifiche nonostante siano usati riferimenti alla classe base o interfaccia Veicolo o Forma.

Si noti bene che questo vale per metodi dichiarati astratti nella classe base astratta o nella interfaccia e che sono poi implementati (o ridefiniti nel caso non siano metodi astratti e si usi l'override) nelle sottoclassi.

Cosa succede invece in caso di metodi specifici aggiunti nella sottoclasse? In questo caso è possibile utilizzare questi metodi aggiuntivi solo quando si lavora con un riferimento all'oggetto della sottoclasse specifica e non quando si lavora con un riferimento alla classe base o interfaccia.

Si consideri l'esempio mostrato in Fig. 8. Supponiamo di avere una classe Veicolo con un metodo muovi() e vai() e una sottoclasse Auto che estende la classe Veicolo e definisce un metodo accelera().

Possiamo creare un array di oggetti Veicolo che contiene un oggetto di ogni sottoclasse.

```
public class Veicolo {
    public void muovi() {
        System.out.println("Il veicolo si muove.");
    }
    public void vai() {
        System.out.println("Il veicolo va.");
    }
}
public class Auto extends Veicolo {
     public void muovi() {
        System.out.println(«L'auto si muove.»);
     }
     public void vai() {
        System.out.println(«L'auto va.»);
     }
     public void accelera() {
        System.out.println("L'auto accelera.");
    }
}
public class Main {
    public static void main(String[] args) {
        Veicolo[] veicoli = new Veicolo[2];
        veicoli[0] = new Veicolo();
        veicoli[1] = new Auto();

        for (Veicolo veicolo : veicoli) {
               veicolo.muovi();       //ok
               veicolo.vai();         //ok
               veicolo.accelera();    //!!!
        }
   }
}
```

Fig. 8: *Polimorfismo e invocazione metodi aggiuntivi delle sottoclassi*

Nell'array veicoli abbiamo un oggetto di tipo Veicolo e un oggetto di tipo Auto, quindi quando chiamiamo il metodo muovi() e vai() su ciascun oggetto dell'array, viene eseguita l'implementazione appropriata del metodo per la classe di ogni oggetto.

Tuttavia, quando si tenta di chiamare il metodo accelera() su un oggetto di tipo Veicolo (che è presente nell'array in posizione 0), si riceverà un errore di compilazione poiché la classe Veicolo non ha un metodo accelera().

Quando invece si tenta di invocare il metodo accelera() sull'oggetto Auto non si ha errore di compilazione.

Tuttavia il metodo non è lo stesso disponibile in quanto si usa per l'invocazione un riferimento alla classe base (ripeto, questo è possibile perché un oggetto della classe derivata è anche un oggetto della classe base e permette per i metodi della classe base l'invocazione della specifica implementazione nella classe derivata).

Ricapitolando…

In generale nel polimorfismo l'invocazione specifica del metodo dipende dal tipo di riferimento usato per l'invocazione del metodo.

Se la classe effettivamente istanziata è la sottoclasse (sebbene la logica sia basata su riferimento alla classe base -astratta o interfaccia-) allora come visto l'invocazione è sul metodo specifico implementato nella sottoclasse, con i vantaggi che abbiamo visto. Inoltre abbiamo visto che se una sottoclasse definisce metodi aggiuntivi rispetto alla classe base, la raggiungibilità o meno di questi metodi dipende dal tipo di riferimento che viene usato per accedere all'oggetto.

A questo riguardo, in generale e in riferimento alla Fig. 9 si possono distinguere i seguenti sottocasi:

1. Si ha un riferimento alla classe base e si fa la new di una classe base: in questo caso, il tipo del riferimento è Veicolo, quindi è possibile accedere solo ai metodi definiti nella classe base, che è l'unica in gioco. I metodi aggiuntivi definiti nella sottoclasse non sono accessibili tramite questo riferimento (anzi si ha errore di compilazione) in quanto si è istanziato un oggetto di tipo Veicolo e si usa un riferimento ad esso.
2. Si ha un riferimento alla sottoclasse e si fa la new di una sottoclasse: in questo caso, il tipo del riferimento è Auto, quindi è possibile accedere sia ai metodi definiti nella classe base che ai metodi aggiuntivi definiti nella sottoclasse. Per i primi, se ridefiniti, vengono eseguiti le implementazioni dei metodi nella classe Auto.
3. Si ha un riferimento alla classe base e si fa la new di una sottoclasse: in questo caso, il tipo del riferimento è Veicolo, ma l'oggetto effettivo creato è di tipo Auto. In questo caso, in virtù del late binding i metodi muovi e vai che sono invocati sono quelli della sottoclasse Auto (non della classe Veicolo) in virtù del late binding e della new fatta sulla classe Auto. Il metodo accelera() non è disponibile in quanto si usa un riferimento alla classe Veicolo per cui quel metodo non è definito.

```
…
Veicolo v = new Veicolo();
v.muovi();      // invoca il metodo muovi() della classe Veicolo
v.vai();        // invoca il metodo vai() della classe Veicolo
v.accelera();   // non è possibile, perché il metodo non è definito
                // nella classe Veicolo (errore di compilazione)
Auto a = new Auto();
a.muovi();      // invoca il metodo muovi() della classe Auto
a.vai();        // invoca il metodo vai() della classe Auto
a.accelera();   // invoca il metodo aggiuntivo definito nella classe Auto

Veicolo v = new Auto();
v.muovi();      // invoca il metodo muovi() della classe Auto
v.vai();        // invoca il metodo vai() della classe Auto
v.accelera();   // metodo non disponibile, perché il metodo non è definito
                // nella classe Veicolo
….
```

Fig. 9: *Sottocasi di invocazione dei metodi delle sottoclassi nel polimorfismo*

In generale, il polimorfismo consente di usare un'istanza di una sottoclasse come se fosse un'istanza della classe base, ma con l'aggiunta della flessibilità di accedere ai metodi della classe base ridefiniti nella sottoclasse, e solo per questi metodi (non sono inclusi metodi specifici della sottoclasse).

Questi sono invocabili solo se se il tipo del riferimento lo consente (deve essere un riferimento alla sottoclasse).

Riassumendo: Il tipo del riferimento determina quello che si può fare: possiamo invocare solo i metodi definiti nella classe a cui il riferimento appartiene - Il tipo dell'istanza determina cosa viene effettivamente fatto: viene invocato il metodo definito nella classe a cui l'istanza appartiene (polimorfismo).

Questo si vede soprattutto nell'ultimo caso dove si ha la dichiarazione di un riferimento alla classe base e si istanzia una classe derivata da tale classe base.

In definitiva, l'utilizzo di polimorfismo e ereditarietà in Java consente di scrivere codice che può essere facilmente esteso e modificato in futuro senza dover riscrivere il codice esistente.

Ciò aumenta la flessibilità e la manutenibilità del codice, consentendo di gestire meglio i cambiamenti e le evoluzioni del programma nel tempo.

4. Eccezioni

Nel linguaggio Java, un'eccezione (exception) rappresenta un evento anomalo che si verifica durante l'esecuzione di un programma. Quando qualcosa va storto — ad esempio un tentativo di accesso a un elemento fuori dai limiti di un array, oppure la lettura di un file che non esiste — Java interrompe il normale flusso di esecuzione e lancia un oggetto di tipo Throwable.

Il sistema delle eccezioni in Java è potente e strutturato, pensato per gestire in modo pulito gli errori e migliorare l'affidabilità del codice.

Tutte le eccezioni in Java derivano dalla classe base Throwable, che ha due principali sottoclassi:

- Error: usata dal sistema per errori gravi (es. OutOfMemoryError). Non devono essere catturate o gestite normalmente.
- Exception: rappresenta situazioni recuperabili. La maggior parte delle eccezioni personalizzate deriva da questa classe.

Dentro Exception troviamo:

- Checked exceptions: devono essere obbligatoriamente gestite (es. IOException, SQLException).
- Unchecked exceptions: sono sottoclassi di RuntimeException e possono non essere gestite esplicitamente (es. NullPointerException, ArrayIndexOutOfBoundsException).

Per intercettare e gestire le eccezioni, si utilizza il costrutto try-catch.

```
public class Divisione {
   public static void main(String[] args) {
      try {
         int a = 10;
         int b = 0;
         int risultato = a / b;
         System.out.println("Risultato: " + risultato);
      } catch (ArithmeticException e) {
         System.out.println("Errore: divisione per zero.");
      }
   }
}
```

In questo esempio, il programma intercetta l'eccezione ArithmeticException causata dalla divisione per zero e stampa un messaggio personalizzato.

Il blocco finally, se presente, viene sempre eseguito, sia che si verifichi un'eccezione, sia che non si verifichi.

```
try {
   int[] numeri = {1, 2, 3};
   System.out.println(numeri[3]); // Errore: indice fuori limiti
} catch (ArrayIndexOutOfBoundsException e) {
   System.out.println("Indice non valido!");
} finally {
   System.out.println("Blocco finally eseguito.");
}
```

Il blocco finally è essenziale quando si vogliono rilasciare risorse o eseguire azioni di "pulizia", come:

- chiudere file o connessioni a database,
- liberare memoria,
- ripristinare uno stato precedente del sistema,
- sbloccare risorse concorrenti.

Ad esempio:

```
BufferedReader reader = null;
try {
   reader = new BufferedReader(new FileReader("dati.txt"));
   String linea = reader.readLine();
   System.out.println(linea);
} catch (IOException e) {
   System.out.println("Errore nella lettura del file.");
} finally {
   try {
      if (reader != null) {
         reader.close();
      }
   } catch (IOException e) {
      System.out.println("Errore nella chiusura del file.");
   }
}
```

Anche se durante la lettura del file viene lanciata un'eccezione, il blocco finally si assicura che il file venga comunque chiuso, evitando memory leak o blocchi.

È possibile generare un'eccezione manualmente con la parola chiave throw.

```
public class EsempioThrow {
   public static void checkEta(int eta) {
      if (eta < 18) {
         throw new IllegalArgumentException("Età troppo bassa");
      }
   }

   public static void main(String[] args) {
      checkEta(16);
   }
}
```

Se checkEta riceve un'età inferiore a 18, genera un'eccezione di tipo IllegalArgumentException.

Quando un metodo può generare eccezioni checked, deve dichiararlo esplicitamente con throws.

```
public static void leggiFile(String nomeFile) throws IOException {
   FileReader file = new FileReader(nomeFile);
   BufferedReader reader = new BufferedReader(file);
   System.out.println(reader.readLine());
   reader.close();
}
```

Il chiamante di leggiFile sarà obbligato a gestire l'eccezione, o a sua volta dichiararla.

È possibile definire le proprie eccezioni creando una classe che estende Exception o RuntimeException.

```
class MiaEccezione extends Exception {
   public MiaEccezione(String messaggio) {
      super(messaggio);
   }
}
```

E usarla così:

```
public static void verifica(int valore) throws MiaEccezione {
   if (valore < 0) {
      throw new MiaEccezione("Valore negativo non consentito");
   }
}
```

Anziché scrivere tutto il codice di controllo con if-else, in alcuni casi è più corretto e leggibile lanciare eccezioni personalizzate o standard per segnalare errori gravi o anomali nel flusso del programma.

Quando è preferibile usare throw?

- Quando un metodo non è in grado di portare a termine il suo compito e deve delegare la gestione dell'errore ad altri.
- Quando si vuole mantenere separata la logica di controllo dai dettagli di implementazione.
- Quando l'errore rappresenta una violazione delle regole di business o di integrità del programma.

In particolare, "programmare per eccezioni":

- interrompe immediatamente il flusso in caso di errore,
- permette di gestire l'eccezione a un livello più alto nel programma,
- rende il metodo più semplice e focalizzato sul suo compito principale.

5. Generics

I Generics sono una delle caratteristiche più potenti e importanti del linguaggio Java, introdotti con Java 5. Consentono di scrivere codice più flessibile, sicuro e riutilizzabile, permettendo di operare su oggetti di diversi tipi mantenendo la verifica dei tipi a tempo di compilazione.

L'idea alla base dei generics è di parametrizzare i tipi, in modo simile a come si possono parametrare i metodi o le classi. In pratica, ci permettono di scrivere una singola classe, interfaccia o metodo che possa lavorare con diversi tipi di oggetti in modo sicuro.

Prima dell'introduzione dei generics, le strutture dati come List o Map accettavano oggetti di tipo Object, il che richiedeva il cast esplicito degli oggetti al tipo corretto, esponendo il programma a potenziali errori di runtime.

Si riporta un esempio senza generics (Java < 5):

```
List lista = new ArrayList();
lista.add("Ciao");
String saluto = (String) lista.get(0); // Cast esplicito
```

In questo caso, se qualcuno aggiungesse un oggetto di tipo errato alla lista, il cast fallirebbe a runtime.

Di seguito si mostra lo stesso esempio ma con generics (Java 5+):

```
List<String> lista = new ArrayList<>();
lista.add("Ciao");
String saluto = lista.get(0); // Nessun cast necessario
```

Grazie ai generics, il compilatore assicura che solo String possano essere inseriti nella lista, migliorando la sicurezza del codice.

La sintassi base per l'utilizzo dei Generics in Java viene mostrata nell'esempio seguente:

```
public class Scatola<T> {
   private T contenuto;

   public void set(T contenuto) {
      this.contenuto = contenuto;
   }

   public T get() {
      return contenuto;
   }
}
```

L'uso quindi sarebbe il seguente:

```
Scatola<String> scatolaDiTesto = new Scatola<>();
scatolaDiTesto.set("Testo");
String valore = scatolaDiTesto.get();
```

In questo esempio, T è un tipo generico che verrà sostituito con un tipo concreto (es. String) nel momento dell'uso.

Anche i metodi possono essere parametrizzati con tipi generici, indipendentemente dal fatto che la classe lo sia.

```
public class Utility {
   public static <T> void stampa(T valore) {
      System.out.println(valore);
   }
}
```

La chiamata del metodo quindi sarà la seguente:

```
Utility.stampa("Ciao mondo");
Utility.stampa(123);
```

Per scenari più avanzati, Java offre anche la possibilità di limitare i tipi generici tramite l'uso di vincoli (bounded generics). Ad esempio, se si vuole che un certo tipo generico possa essere solo una sottoclasse di Number, si può usare la sintassi <T extends Number>, utile per classi o metodi che devono eseguire operazioni numeriche.

```
public class Statistiche<T extends Number> {
   private T valore;

   public Statistiche(T valore) {
      this.valore = valore;
   }

   public double raddoppia() {
      return valore.doubleValue() * 2;
   }
}
```

Nell'esempio precedente, solo classi che estendono Number (come Integer, Double, Float, ecc.) possono essere usate.

Inoltre, tramite le cosiddette wildcard (?), è possibile esprimere tipi sconosciuti o flessibili in lettura e scrittura, permettendo ad esempio di accettare qualsiasi lista indipendentemente dal tipo degli elementi, oppure solo quelle contenenti sottotipi o supertypes specifici.

```
public void stampaLista(List<?> lista) {
   for (Object elemento : lista) {
      System.out.println(elemento);
   }
}
```

In particolare:

- <?>: tipo sconosciuto
- <? extends Tipo>: accetta qualsiasi sottotipo di Tipo (covarianza, solo lettura)
- <? super Tipo>: accetta qualsiasi supertipo di Tipo (controvarianza, utile in scrittura)

Un aspetto importante da comprendere è che i generics in Java sono implementati tramite una tecnica chiamata "type erasure", che rimuove le informazioni sui tipi generici a tempo di compilazione.

Questo significa che, a runtime, una List<String> e una List<Integer> sono in realtà la stessa cosa dal punto di vista della JVM. Questo comporta alcune limitazioni, come l'impossibilità di creare array generici o di istanziare direttamente un oggetto di tipo generico con new T().

Nonostante queste restrizioni, l'uso dei generics rappresenta un grande passo avanti in termini di qualità e robustezza del codice. Le collezioni della libreria standard di Java – come ArrayList, HashMap, Queue – sono state tutte riscritte per sfruttare i generics, migliorando notevolmente l'esperienza di programmazione. L'utilizzo corretto dei generics riduce la necessità di controlli manuali sui tipi, semplifica la manutenzione del codice e rende le API più chiare e sicure. Chiunque programmi in Java dovrebbe familiarizzare con questo strumento fondamentale, che è ormai parte integrante della scrittura di codice moderno e professionale nel linguaggio.

Bibliografia

- Il nuovo Java, Claudio De Sio Cesari, Hoepli Informatica
- Programmazione di base e avanzata con Java, Walter Savitch, Pearson