PROGR. A OGGETTI SCHEMI

Concetti di base

CLASSE

NORMALE: definizione e implementazione di un tipo
INTERFACCIA: solo definizione e metodi astratti (no body)
ASTRATTA: definizione + implementazione parziale (presenza metodi astratti)
CLASSE GENERICA: parametrizzata su più tipi (riutilizzabile con più tipi)

ACCESSIBILITÀ DI UNA CLASSE:

- public: accessibile da qualsiasi classe
- default: accessibile da classi nello stesso package o sottoclassi
- protected: solo per classi interne

CLASSE TIPO CONCRETO

```
public class MyClass{
    //MISSION:
    //abstract function:
    //invariante:

    private Tipo variabiledistanza1
    private Tipo variabiledistanza2

    public costruttoreMyClass(){...}

    public Tipo metodo1(Tipo arg1, Tipo arg2, ...){...}
    public Tipo metodo2(Tipo arg1, Tipo arg2, ...){...}
}//end MyClass
```

CLASSE MAIN

```
public class Main{
    public static void main(String[] args){
        //inizializza, per esempio API
        Tipo nome1 = new Tipo();
        Tipo nome1 = new Tipo();

        //esempio uso, magari con try/catch
    }
}//end Main
```

INTERFACCIA

Interfaccia fornisce:

- nome tipo, senza implementazione
- metodi astratti con relativi contratti
- costanti uniche variabili d'istanza ammesse
- strumento per **dependency inversion**

Classe che implementa interfaccia:

- tutti i metodi dell'interfaccia (essendo astratti)
- eventualmente altri metodi + variabili d'istanza

```
public interface Interfaccia{
    //metodi astratti
    Tipo metodo1(Tipo argomento);
    Tipo metodo2(Tipo argomento);
    Tipo metodo3(Tipo argomento);
}

public MyClass implements Interfaccia{
    //eventuali variabili d'istanza
    //implementazione metodi astratti con body
}
```

CLASSE ASTRATTA

Classe con costrutto `abstract` che prevede un'implementazione parziale per la presenza di **metodi astratti**.

```
public abstract class MyClass{
    protected Tipo variabile;
    //costruttore
    public MyClass(Tipo variabile){...}
    //metodi concreti con body
    public void nomeMetodo(Tipo arg){ body; }
    //metodi astratti
    public abstract tipo nomeMetodo();
}//end
```

INTERFACCIA o CLASSE ASTRATTA?

Usare una **classe astratta** quando:

- famiglia di tipi che condividono comportamento
- sottoclassi condividono parte dello stato
- variabili non statiche o final

Usare un'**interfaccia** quando

- sottotipi sono completamente scollegati
- comportamento specifico ma senza implementazione
- sfruttare **multi-ereditarietà**

TIPI

PRIMITIVI: rappresentano valori

- byte, short, int, long, float, double, char, boolean

RIFERIMENTO: rappresentano oggetti mutabili (cambiano stato) o immutabili (stato non cambia). Possono essere shared se sono condivisi in 2 o più variabili.

CONCRETO (classe concreta): può essere istanziato, implementazione completa
ASTRATTO (classe astratta o interfaccia): non può essere istanziato, funge da contratto o base per altri tipi
REALE: classe concreta a cui appartiene l'oggetto
APPARENTE: tipo dichiarato di variabile, definisce i metodi e campi visibili al compilatore. Deve essere uguale o **supertipo** del tipo reale.
SUPERTIPO: classe generale, include operazioni comuni ai sottotipi
SOTTOTIPO: estende o implementa il supertipo,deve rispettare contratto del supertipo

```
int [] a = new int[3];
Object x = a;

//tipo reale, apparente di a: int[]
//tipo reale di x: int[]
//tipo apparente di x: Object

a.length(); //3
a = x; //errore di compilazione
x.length(); //errore di compilazione (Object non ha length())
((int[]) x).length(); //3
```

EREDITARIETÀ e GERARCHIA DI CLASSI

Deve valere il **principio di sostituibilità di Liskov**: sottotipo deve portarsi sostituire al supertipo

- firma: sottotipo deve essere compatibile con supertipo
  - TUTTI metodi astratti presenti e con stessa firma
  - metodi concreti non sovrascritti ereditati automaticamente
- metodi: sottotipo deve chiedere di meno e promettere di più
  - sottotipo più generale: meno eccezione o in meno casi PRE_super => PRE_sottotipo
  - sottotipo promette di più: (PRE_super & POST_sotto) => POST_super
    esempio. post_super val >0, post_sotto > 1
- proprietà: sottotipo mantiene proprietà supertipo
  - proprietà invariante per oggetti
  - proprietà evolutive

APPARENTE: **supertipo** (meno funzioni) → REALE: **sottotipo** (più funzioni, richiede meno vincoli, può sollevare meno eccezioni)

```
List<Student> lista = new ArrayList<Student>(); //apparente: List, reale: ArrayList
```

GERARCHIA DI CLASSI

```
public class Sottotipo extends Supertipo{
    //variabili d'istanza super
    //variabili d'istanza sottotipo
    //@Override metodi super (obbligatiori se astratti)
    //metodi sottotipo
}
```

POLIMORFISMO

- metodi: posso instradare verso metodi diversi in una classe in base alle firme
- oggetti: istanza supertipo castata in istanza sottotipo

principio liskov

- rispettato: uso gerarchia + ereditarietà
- non rispettato: delega o associazione

TIPI DATI ASTRATTI (adt)

obiettivo degli adt è creare una struttura che ricacalchi il problema

Un dato astratto è composto da:

- rappresentazione: strutture dati, componenti
- operazioni: metodi dell'adt

Viene definito dal suo comportamento

variabili d'istanza (private) e metodi **non** devono essere **statici**

SPECIFICAZIONE:

- STATO ASTRATTO: valori possibili adt (es. matrice bidimensionale di float)
- PROTOCOLLO: operazioni su valori, vale anche sequenza di chiamate di operazioni, con contratti riferiti allo stato astratto

esempio. ADT = insieme di interi. Metodo ha delle POST condizioni su eccezione

- `//solleva EmptyIntSetException se array ha 0 elementi` : NON VA BENE, riferito a implementazione
- `//solleva EmptyIntSetException se insieme è vuoto` : OTTIMO, riferito a stato astratto

MISSION: cosa sa (stato) e cosa può fare (transizioni di stato)

INVARIANTE: "matrice valida m definita come m è NxM con N,M > 0"

- indipendente da implementazione
- funge da pre per tutti i metodi (tranne per costruttore)
- vale per tutti gli stati

INVARIANTE DI RAPPRESENTAZIONE:

- legato alla funzione di astrazione f: dato astratto → stato concreto
- descritto nella classe

TIPO GENERICO

garantisce type safety a compile-time

```
public class Box<T> {
    private T value; //var d'ist.
    public void set(T value) {this.value = value;}
    public T get() {return value;}
}

//uso
Box<Integer> b = new Box<>();
b.set(10);
Integer x = b.get();
```

Anche con interfaccia

```
public interface Repository<T> {
    void save(T obj);
    T findById(int id);
}

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

METODI

accessibilità dei metodi:

- **PUBLIC**: da qualsiasi classe → API, servizi, operazioni ADT
- **PROTECTED**: nel package o da sottoclassi esterne → per design pattern come
- **DEFAULT**: solo nel package → collaborazioni tra classi nel package, logica condivisa
- **PRIVATE**: solo all'interno della classe → per metodi di supporto, logica e uso interni

come si chiama un metodo:
espressione0.metodo(espressione1, espressione2, ...)

- parte da oggetto e poi valuta espressione 1, 2, ...

- se non esiste oggetto solleva NullPointerException

```
public Tipo NomeMetodo(Tipo argomento) throws Tipo exception{
    //PRE: pre-condizioni affinchè il metodo ritorni valore atteso, meglio se assenti
    //POST: post-condizioni, eccezioni sollevate
    //MODIFIES: elenco variabili d'istanza modificate

    body;
}
```

OVERRIDE

- ridefinizione di un metodo ereditato da supertipo o interfaccia
- firma identica (nome, parametri, tipo ritorno)

```
class Animale{
    public void faiVerso(){System.out.println("Verso generico");}
}

class Cane extends Animale{
    @Override
    public void faiVerso(){System.out.println("Bau Bau");}
}
```

OVERLOAD

- definizione di più metodi con stesso nome ma firme diverse (diversi tipi o numero di parametri)

```
class Calcolatrice{
    public int somma(int a, int b){ return a + b; }
    public double somma(double a, double b){ return a + b; }
    public int somma(int a, int b, int c){ return a + b + c; }
}
```

CONTRATTO

- pre-condizioni: cosa deve essere vero prima di chiamare il metodo
- post-condizioni: cosa deve essere vero dopo l'esecuzione del metodo

ECCEZIONI

Stato di un programma: combinazione di variabili d'istanza e stack di chiamate attive

tipi di errori di programazione:

- fault: bug nel codice (es. divisione per 0)
- failure: comportamento anomalo in fase di esecuzione (es. file non trovato)
- error: problema grave che impedisce l'esecuzione (es. memoria insufficiente)
- design mistake: errore concettuale nel design (es. violazione del principio di sostituibilità di Liskov)

Eccezioni: servono a gestire situazioni eccezionali

- non infliscono il flusso normale del programma
- non influenzano il set dei risultati attesi
- chiamante può decidere se gestirle o propagare
- esplicita rappresentazione di errori

ECCEZIONI CHECKED

- verificate a compile-time
- devono essere dichiarate nel metodo con `throws` o gestite con `try-catch`

```
public class MyCheckedException extends Exception{
    public MyCheckedException(){
        super();
    }
    public MyCheckedException(String message){
        super(message);
    }
}
```

ECCEZIONI UNCHECKED

- verificate a runtime
- non è obbligatorio dichiararle o gestirle
- rappresentano violazioni di precondizioni, logicamente non recuperabili

```
public class MyUncheckedException extends RuntimeException{
    public MyUncheckedException(){
        super();
    }
    public MyUncheckedException(String message){
        super(message);
    }
}
```

ASSERZIONI

- condizioni logiche verificate durante l'esecuzione
- usate per validare stati interni del programma

```
assert condizione_logica : "Messaggio di errore se l'asserzione fallisce";
//esempio
assert x > 0 : "x deve essere positivo";
```

ITERATORI

- oggetti che permettono di attraversare una collezione senza esporre la sua rappresentazione interna
- possono essere definiti anche all'interno di una classe come classi interne

```
public class MyCollection{
    private Tipo[] elementi;
    private int size;

    public Iterator<Tipo> iterator(){
        return new ResettableMyIterator();
    }


    private class ResettableMyIterator implements Iterator<Tipo>{
        private int currentIndex = 0; //variabile indice
        @Override public boolean hasNext(){}
        @Override public Tipo next(){}
        @Override public void remove(){}
        public void reset(){}
}
```

ENUM

- insieme finito di costanti statiche e immutabili

```
public enum MyEnum{
    ENUM1("Valore1"),
    ENUM2("Valore2"),
    ENUM3("Valore3");

    private String valore;

    private MyEnum(String valore){this.valore = valore;}
    public String getValore(){return valore;}
}
```

DESIGN PATTERN

- soluzioni riutilizzabili a problemi comuni di progettazione software

SINGLETON

- garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa

```
public class Singleton{
    private static Singleton instance = null;

    private Singleton(){ //costruttore privato
    }

    public static Singleton getInstance(){
        if(instance == null){
            instance = new Singleton();
        }
        return instance;
    }
}
```

FACTORY

- fornisce un'interfaccia per creare oggetti in una superclasse, ma permette alle sottoclassi di alterare il tipo di oggetti che verranno creati

```java
//interfaccia prodotto
public interface Veicolo {
    void muovi();
}

//classi concrete prodotto
public class Auto implements Veicolo {
    @Override public void muovi() { System.out.println("L'auto si muove");}
}

public class Moto implements Veicolo {
    @Override public void muovi() { System.out.println("La moto si muove");}
}

//factory
public class VeicoloFactory {
    public static Veicolo creaVeicolo(String tipo) {
        if (tipo.equalsIgnoreCase("auto")) {
            return new Auto();
        } else if (tipo.equalsIgnoreCase("moto")) {
            return new Moto();
        }
        return null;
    }
}

//uso della factory
Veicolo veicolo1 = VeicoloFactory.creaVeicolo("auto");
veicolo1.muovi(); // Output: L'auto si muove
Veicolo veicolo2 = VeicoloFactory.creaVeicolo("moto");
veicolo2.muovi(); // Output: La moto si muove
```

BUILDER

provvede flessibilità quando inizializzi un tipo con molti attributi

- possibilità di campi opzioni e non
- generalmente classe interna

```java
//classe prodotto
public class Persona{
    private String nome;
    private String cognome;
    private int età;
    private String email;

    private Persona(PersonaBuilder builder){
        this.nome = builder.nome;
        this.cognome = builder.cognome;
        this.età = builder.età;
        this.email = builder.email;
    }

    //classe builder interna
    public static class PersonaBuilder{
        public String nome;
        public String cognome;
        public int età;
        public String email;

        //costruttore con campi obbligatori
        public PersonaBuilder(String nome, String cognome){
            this.nome = nome;
            this.cognome = cognome;
        }

        //metodi per campi opzionali
        public PersonaBuilder età(int età){
            this.età = età;
            return this;
        }

        public PersonaBuilder email(String email){
            this.email = email;
            return this;
        }
        //metodo build essenziale per costruire
        public Persona build(){
            return new Persona(this);
        }
    }
}

//uso del builder
Persona persona = new Persona.PersonaBuilder("Mario", "Rossi")
                        .età(30)
                        .email("mario.rossi@example.com")
                        .build();
```

OBSERVER

- serve a disaccopiare il soggetto osservato dagli osservatori

```java
//interfaccia osservatore
public interface Observer {
    void update(Object data);
}
//interfaccia soggetto osservabile
public interface Observable {
    void addObserver(Observer o);
    void removeObserver(Observer o);
    void notifyObservers(Object data);
}
//classe concreta soggetto osservabile
public class Agenzia implements Observable {
    //lista di osservatori
    private List<Observer> observers = new ArrayList<>();
    private String msg;

    @Override
    public void addObserver(Observer o) {observers.add(o);}

    @Override
    public void removeObserver(Observer o) {observers.remove(o);}

    @Override
    public void notifyObservers(Object data) {
        for (Observer o : observers) {
            o.update(data);
        }
    }
    //metodo con stato che vogliamo comunicare
    public void setMsg(String notizia) {
        this.msg = notizia;
        notifyObservers(notizia);
    }

//classe osservatore
public class Quotidiano implements Observer {
    private String nome;
    //costruttore
    public Quotidiano(String nome) {this.nome = nome;}

    //metodo update
    @Override
    public void update(Object news) {
        System.out.println(nome + ":" + news);
        }
    }
}

public class client{
    public static void main(String[] args){
```

```
        Agenzia agenzia = new Agenzia();

        Quotidiano q1 = new Quotidiano("Giornale1");
        Quotidiano q2 = new Quotidiano("Giornale2");

        agenzia.addObserver(q1);
        agenzia.addObserver(q2);

        agenzia.setMsg("Notizia importante!");
    }
}
```

DECORATOR

- aggiunge responsabilità/comportamenti dinamicamente
- basandomi su un'interfaccia comune, creo una classe astratta decorator che la implementa e contiene un riferimento all'oggetto da decorare

```java
//interfaccia componente I
public interface Drink {
    String getDescription();
    double costo();
}

//classe concreta componente O implementa I
public class Espresso implements Drink {
    @Override
    public String getDescription() { return "Espresso"; }
    @Override
    public double costo() { return 1.50; }
}

//classe decoratore astratta D implementa I
public abstract class DrinkDecorator implements Drink {
    protected Drink drink;
    public DrinkDecorator(Drink drink) {this.drink = drink;}
}

//classe decoratore concreta estende D
public class Latte extends DrinkDecorator {
    public Latte(Drink drink) {super(drink);}
    @Override
    public String getDescription() {return drink.getDescription() + ", Latte";}
    @Override
    public double costo() {return drink.costo() + 0.50;}
}

public class Choco extends DrinkDecorator {
    public Choco(Drink drink) {super(drink);}
    @Override
    public String getDescription() {return drink.getDescription() + ", Choco";}
    @Override
    public double costo() {return drink.costo() + 0.70;}
}

//uso del decorator
Drink myDrink = new Espresso();
myDrink = new Latte(myDrink);
myDrink = new Choco(myDrink);

System.out.println(myDrink.getDescription() + " $" + myDrink.costo());
>>> Espresso, Latte, Choco $2.70
```

COMPOSITE

- compone oggetti in strutture ad albero per rappresentare gerarchie parte-tutto

```java
//interfaccia componente
public interface FileSysComp{
    void stampa();
    int getSize();
}

//classe foglia
public class File implements FileSysComp{
    private String nome;
    private int size;

    //costruttore
    public File(String nome, int size){...}

    @Override public void stampa(){stampa nome + dim;}
    @Override public int getSize(){return size;}
}

//classe composita
public class Directory implements FileSysComp{
    private String nome;
    private List<FileSysComp> contenuto = new ArrayList<>();

    //costruttore
    public Directory(String nome){...}

    public void add(FileSysComp componente){contenuto.add(componente);}

    @Override
    public void stampa(){ stampa el. contenuto};}

    @Override
    public int getSize(){somma dim. contenuto;}
}

//uso del composite
FileSysComp f1 = new File("a.txt", 1);
FileSysComp f2 = new File("b.txt", 200);
FileSysComp dir1 = new Directory("dir1");
((Directory) dir1).add(f1);
((Directory) dir1).add(f2);

}
```

BRIDGE

- disaccoppiare astrazione da implementazione

```java
//interfaccia implementazione
public interface Dispositivo{
    void accendi();
    void spegni();
    void setVol(int vol);
    }

//classe concreta implementazione
public class TV implements Dispositivo{
    @Override public void accendi(){...}
    @Override public void spegni(){...}
    @Override public void setVol(int vol){...}
}

//classe astratta astrazione
public abstract class Telecomando{
    protected Dispositivo dispositivo;
    public Telecomando(Dispositivo dispositivo){...}
    public abstract void accendi();
    public abstract void spegni();
    public abstract void alzaVolume();
}

//classe concreta astrazione
public class TelecomandoBase extends Telecomando{
    public TelecomandoBase(Dispositivo dispositivo){super(dispositivo);}
    @Override public void accendi(){dispositivo.accendi();}
    @Override public void spegni(){dispositivo.spegni();}
    @Override public void alzaVolume(){dispositivo.setVol(...);}
}

//uso del bridge
Dispositivo tv = new TV();
Telecomando telecomando = new TelecomandoBase(tv);
telecomando.accendi();
```

VISITOR

aggiunge nuove funzionalità a un oggetto, implementa separazione dati e operazioni

```java
//interfaccia visitor, implementa metodo visit per ogni tipo
public interface ShapeVisitor{
    void visit(Cerchio c);
    void visit(Rettangolo r);
}

//interfaccia elementi
public interface Shape{
    void accept(ShapeVisitor visitor);
}

//elementi concreti
public class Cerchio implements Shape {
    private double raggio;
    public Cerchio(double raggio) {this.raggio = raggio;}
    public double getRaggio() {return raggio;}

    @Override
    public void accept(ShapeVisitor visitor) {visitor.visit(this);}
}

public class Rettangolo implements Shape {
    private double base;
    private double altezza;
    public Rettangolo(double base, double altezza) { ...}
    public double getBase() {...}
    public double getAltezza() {..}

    @Override
    public void accept(ShapeVisitor visitor) {
        visitor.visit(this);
    }
}

//visitor concreti, ovvero implementazioni operazioni
public class AreaVisitor implements ShapeVisitor {
    @Override
    public void visit(Cerchio c) {formula area; stampa area;}
    @Override
    public void visit(Rettangolo r) {formula area; stampa area;}
}

public class PrintVisitor implements ShapeVisitor {
    @Override
    public void visit(Cerchio c) {print get.raggio();}
    @Override
    public void visit(Rettangolo r) {stampa;}
}

//uso all'interno del main(String[] args)
```

```
Shape[] shapes = {
            new Cerchio(3),
            new Rettangolo(4, 5)
        };

        ShapeVisitor areaVisitor = new AreaVisitor();
        ShapeVisitor printVisitor = new PrintVisitor();

        for (Shape s : shapes) {
            s.accept(areaVisitor);
            s.accept(printVisitor);
        }
```

STRATEGY

definisce famiglia di algoritmi incapsulati in classi separate e intercambiali in runtime

```java
//interfaccia
public interface SpedizioneStrategy {
    double costo(double peso);
}

//classi concrete che implementano algoritmo

public class Standard implements SpedizioneStrategy {
    @Override public double costo(double peso) {return peso * 2;}
}

public class Express implements SpedizioneStrategy {
    @Override public double costo(double peso) {return peso * 3;}
}

public class Internazionale implements SpedizioneStrategy {
    @Override public double costo(double peso) {return peso * 4;}
}

//classe che usa strategia
public class SetSpedizione{ {
    private SpedizioneStrategy tipologia; //strategia
    public void setTipologia(SpedizioneStrategy tipologia) {
        this.tipologia = tipologia;
    }

    public double costo(double peso) {
        return tipologia.costo(peso);
    }
}

//uso nel main(String[] args)
//inizializzo classe che usa strategia
SetSpedizione spedizione = new SetSpedizione();
//imposto e uso diverse strategie, basta cambiare classe nell'argomento
spedizione.setTipologia(new Standard());
System.out.println(spedizione.costo(5));
>> 10.0
```