

TIPI DI CLASSI

Classi Innestate o Inner

```
public class Outer
{
    private String messaggio = "Nella classe ";
    private void stampaMessaggio()
    {
        System.out.println(messaggio + "Esterna");
    }
    /* la classe interna accede in maniera naturale ai membri
    della classe che la contiene */
    public class Inner // classe interna
    {
        public void metodo()
        {
            System.out.println(messaggio + "Interna");
        }
        public void chiamaMetodo()
        {
            stampaMessaggio();
        }
        . . .
    }
    . . .
}
```

Sono definite
all'interno di un'altra
classe

- Anche in un metodo

Caratteristiche

- Hanno accesso alle variabili d'istanza

Trovano impiego nelle
GUI

Classi Anonime

- Sono delle classi innestate senza “nome”
- Caratteristiche
 - Non hanno costruttore
 - Estendono un’ altra classe
 - Si dichiarano quando si istanziano

```
public class Outer
{
    private String messaggio = "Nella classe ";
    private void stampaMessaggio()
    {
        System.out.println(messaggio+"Esterna");
    }
    //Definizione della classe anonima e sua istanza
    ClasseEsistente ca = new ClasseEsistente()
    {
        public void metodo()
        {
            System.out.println(messaggio+"Interna");
        }
    }; //Notare il ";"
    . . .
}
//Superclasse della classe anonima
public class ClasseEsistente
{
    . . .
}
```

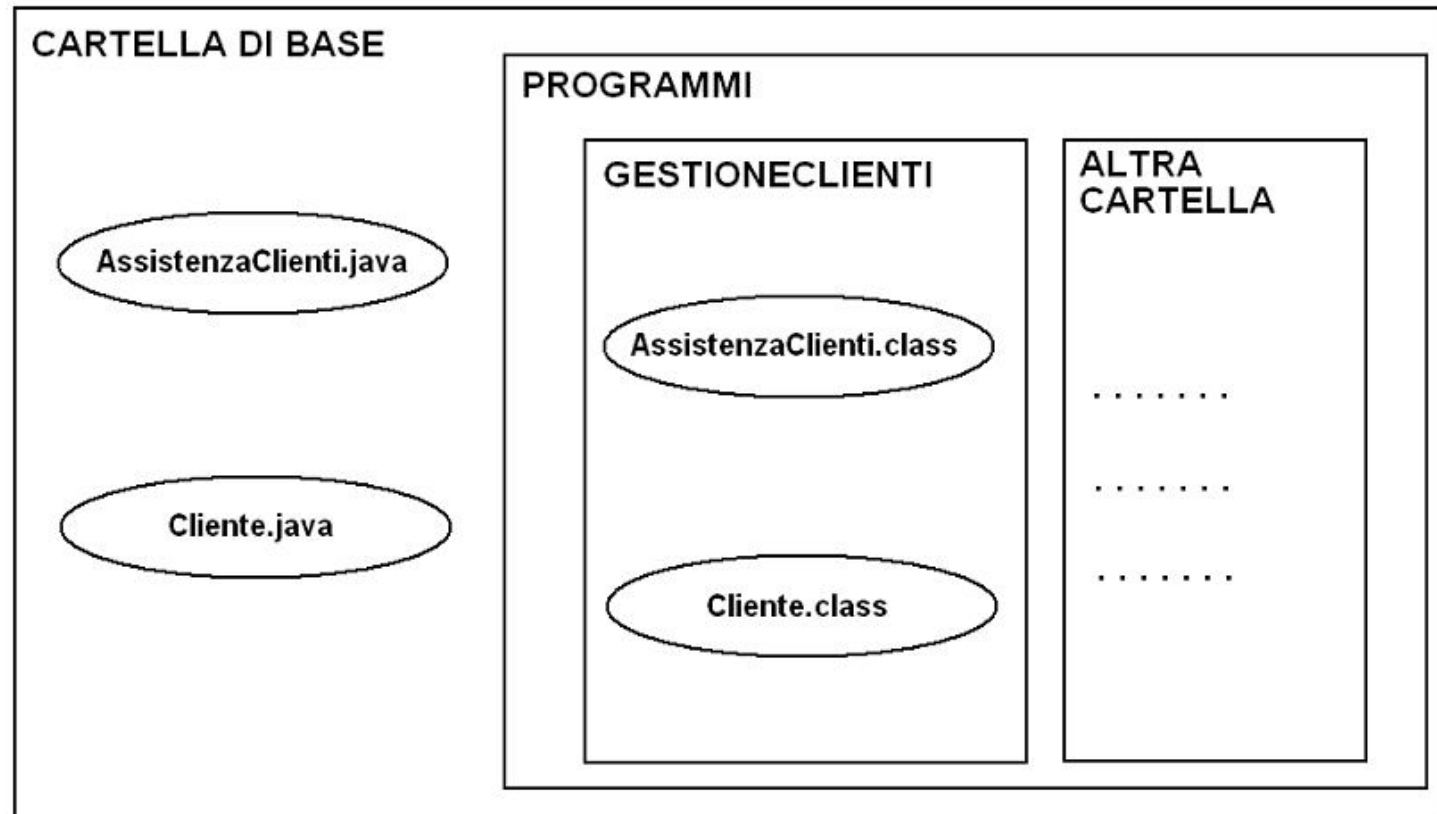
MODIFICATORI

Modificatori

MODIFICATORE	CLASSE	ATTRIBUTO	METODO	COSTRUTTORE	BLOCCO DI CODICE
<code>public</code>	sì	sì	sì	sì	no
<code>protected</code>	no	sì	sì	sì	no
<code>(default)</code>	sì	sì	sì	sì	sì
<code>private</code>	no	sì	sì	sì	no
<code>abstract</code>	sì	no	sì	no	no
<code>final</code>	sì	sì	sì	no	no
<code>native</code>	no	no	sì	no	no
<code>static</code>	no	sì	sì	no	sì
<code>strictfp</code>	sì	no	sì	no	no
<code>synchronized</code>	no	no	sì	no	no
<code>transient</code>	no	sì	no	no	no

Package

```
package programmi.gestioneClienti;
public class AssistenzaClienti
{
    . . . . .
}
```



Modificatori di accesso

- `public`, (Default), `protected` e `private`

MODIFICATORE	STESSA CLASSE	STESSO PACKAGE	SOTTOCLASSE	DAPPERTUTTO
<code>public</code>	sì	sì	sì	sì
<code>protected</code>	sì	sì	sì	no
(default)	sì	sì	no	no
<code>private</code>	sì	no	no	no

Modificatore final

- **Caratteristiche:**
 - una variabile dichiarata final diviene una costante
 - un metodo dichiarato final non può essere riscritto in una sottoclasse (non è possibile applicare l'override)
 - una classe dichiarata final non può essere estesa
- Posso dichiarare final anche parametri e variabili locali di metodi

Modificatore static

- Se dichiaro del codice static questo è:
 - “condiviso da tutte le istanze della classe”
 - oppure diciamo solo “della classe”
- Per accedere a membri statici

```
NomeClasse.nomeMembro
```

- Esempio metodo

```
Math.sqrt(numero)
```

- Un metodo statico non può accedere alle variabili di istanza senza referenziarle

Variabili statiche

- Contiamo le istanze

```
public class Counter {
    private static int counter = 0;
    private int number;
    public Counter() {
        counter++;
        setNumber(counter);
    }
    public void setNumber(int number) {
        this.number = number;
    }
    public int getNumber() {
        return number;
    }
}
```

```
Counter c1 = new Counter();
```

```
Counter c2 = new Counter();
```

Inizializzatori statici

- Blocchi di codice definiti nella classe
 - È chiamato quando la classe (non gli oggetti) è caricata in memoria

```
public class EsempioStatico
{
    private static int a = 10;
    public EsempioStatico()
    {
        a += 10;
    }
    static
    {
        System.out.println("valore statico = " + a);
    }
}
```

```
EsempioStatico ogg = new EsempioStatico();
```

```
valore statico = 10
```

Modificatore abstract

- **Metodi astratti**
 - Non dichiarano il corpo del metodo
 - Lo deve dichiarare la sottoclasse
 - Esistono solo nelle classi astratte
- **Classi Astratte**
 - Non si possono istanziare

```
public abstract class Pittore
{
    . . .
    public abstract void dipingiQuadro();
    . . .
}
```

Interfacce

- Un interfaccia possiede :
 - tutti i metodi dichiarati public e abstract
 - tutte le variabili dichiarate public, static e final

```
public interface Saluto
{
    String CIAO = "Ciao";
    String BUONGIORNO = "Buongiorno";
    . . .
    void saluta();
}
```

```
public class SalutoImpl implements Saluto
{
    public void saluta()
    {
        System.out.println(CIAO);
    }
}
```

Ereditarietà Multipla

- Java simula l' ereditarietà multipla mediante le interfacce

```
public class MiaApplet extends Applet implements
MouseListener, Runnable
{
    . . .
}
```

Conversione di tipo

- Esempio
 - A e B sono due interfacce
 - B estende l' interfaccia A
 - C è una classe che implementa B
- Posso scrivere

```
B b = new C();
A a = b;
```

Differenze tra Classi Astratte e Interfacce

- Sia interfacce che classi astratte obbligano ad implementare dei comportamenti nelle sottoclassi
- Nelle classi astratte posso mettere del codice per le sottoclassi
- L' ereditarietà multipla si simula solo con le interfacce
- L' uso di entrambe le soluzioni è legato al polimorfismo

Enumerazioni

- un accenno

```
public enum MiaEnumerazione {  
    UNO, DUE, TRE;  
}
```

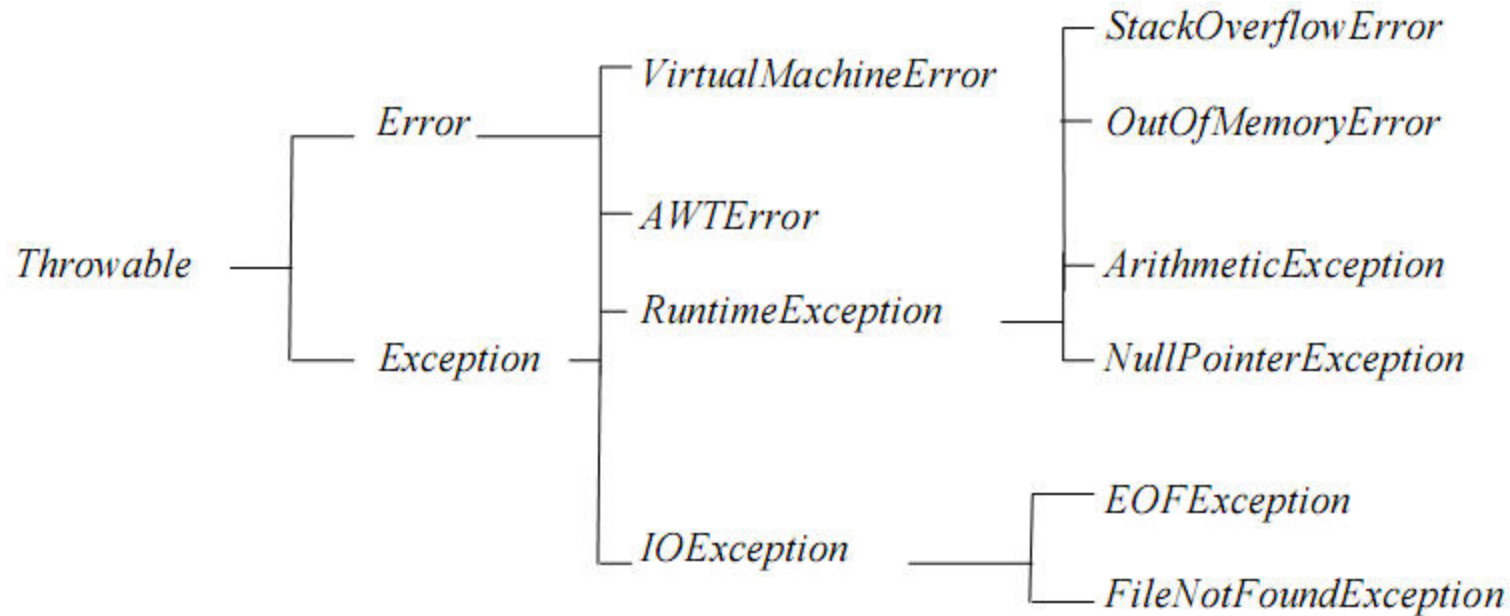
- Tutte le istanze di un'enumerazione sono implicitamente dichiarate **public, static** e **final**

```
System.out.println(MiaEnumerazione.UNO);
```

Eccezioni, errori ed asserzioni

- Le **eccezioni** sono delle situazioni impreviste che il flusso di una applicazione può incontrare
 - Gestite mediante try, catch, throws, throw, finally
 - Concetto implementato mediante **Exception**
- Gli **errori** sono delle situazioni impreviste che non dipendono dal programmatore.
 - Non sono gestibili
 - Implementate mediante **Error**
- Le asserzioni sono condizioni che devono essere verificate perché una parte di codice sia corretta
 - Gestite dalla parola chiave assert

Classi per eccezioni ed errori



Gestione delle Eccezioni

```
public class Ecc1 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        int c = a/b;
        System.out.println(c);
    }
}
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
at Ecc1.main(Ecc1.java:6)

- Viene lanciata l'eccezione ma il programma è terminato

Catturare l'eccezione

```
public class Ecc2 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
    }
}
```

- Stampare il contenuto di una eccezione

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    exc.printStackTrace();
}
```

Eccezioni generiche

- Se gestisco l'eccezione “sbagliata” il programma termina

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (NullPointerException exc) {
    exc.printStackTrace();
}
```

- Posso usare il polimorfismo

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

Gestire più eccezioni

```
int a = 10;
int b = 0;
try {
    int c = a/b;
    System.out.println(c);
}
catch (ArithmeticException exc) {
    System.out.println("Divisione per zero...");
}
catch (NullPointerException exc) {
    System.out.println("Reference nullo...");
}
catch (Exception exc) {
    exc.printStackTrace();
}
```

- La prima che corrisponde viene eseguita
 - Conta l'ordine in cui le scriviamo

finally

```
public class Ecc4 {
    public static void main(String args[]) {
        int a = 10;
        int b = 0;
        try {
            int c = a/b;
            System.out.println(c);
        }
        catch (ArithmeticException exc) {
            System.out.println("Divisione per zero...");
        }
        catch (Exception exc) {
            exc.printStackTrace();
        }
        finally {
            System.out.println("Tentativo di
operazione");
        }
    }
}
```

- Viene eseguito sempre
 - Serve per garantire il funzionamento di un blocco di codice

Creare eccezioni

- È possibile creare delle eccezioni
 - Si estende la classe Exception

```
public class PrenotazioneException extends Exception {
    public PrenotazioneException() {
        // Il costruttore di Exception chiamato inizializza la
        // variabile privata message
        super("Problema con la prenotazione");
    }
    public String toString() {
        return getMessage() + ": posti esauriti!";
    }
}
```

- Devo dire a java quando lanciare questa eccezione

Lanciare eccezioni

- Creare l'oggetto eccezione
- Lanciarla con throw - sintassi:

```
PrenotazioneException exc = new PrenotazioneException();
throw exc;
```

- Esempio

```
try {
    //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
    }
    //istruzione eseguita
    // se non viene lanciata l'eccezione
    postiDisponibili--;
}
catch (PrenotazioneException exc) {
    System.out.println(exc.toString());
}
```

Esempio di propagazione

```
public class Botteghino {
    private int postiDisponibili;

    public Botteghino() {
        postiDisponibili = 100;
    }

    public void prenota() {
        try {
            //controllo sulla disponibilità dei posti
            if (postiDisponibili == 0) {
                //lancio dell'eccezione
                throw new PrenotazioneException();
            }
            //metodo che realizza la prenotazione
            // se non viene lanciata l'eccezione
            postiDisponibili--;
        }
        catch (PrenotazioneException exc){
            System.out.println(exc.toString());
        }
    }
}
```

Spesso dove viene generata una eccezione non si sa come gestirla

```
public class GestorePrenotazioni {
    public static void main(String [] args) {
        Botteghino botteghino = new Botteghino();
        for (int i = 1; i <= 101; ++i){
            botteghino.prenota();
            System.out.println("Prenotato posto n° " + i);
        }
    }
}
```

Esempio di propagazione

```
public class GestorePrenotazioni {
    public static void main(String [] args) {
        Botteghino botteghino = new Botteghino();
        try {
            for (int i = 1; i <= 101; ++i){
                botteghino.prenota();
                System.out.println("Prenotato posto n° " + i);
            }
        }
        catch (PrenotazioneException exc) {
            System.out.println(exc.toString());
        }
    }
}
```

Dichiaro il metodo per lanciare eccezioni

```
public void prenota() throws PrelievoException {
    //controllo sulla disponibilità dei posti
    if (postiDisponibili == 0) {
        //lancio dell'eccezione
        throw new PrenotazioneException();
    }
    //metodo che realizza la prenotazione
    // se non viene lanciata l'eccezione
    postiDisponibili--;
}
```

Eccezioni ed override

- Riscrivendo un metodo
 - Non posso aggiungere clausole throws
 - Se c'è la devo includere
 - Se c'è posso specificare come eccezione una sottoclasse dell'eccezione dichiarata

```
public class ClasseBase {
    public void metodo() throws java.io.IOException { }
}

class SottoClasseCorretta1 extends ClasseBase {
    public void metodo() throws java.io.IOException {}
}

class SottoClasseCorretta2 extends ClasseBase {
    public void metodo() throws
java.io.FileNotFoundException {}
}

class SottoClasseCorretta3 extends ClasseBase {
    public void metodo() {}
}

class SottoClasseScorretta extends ClasseBase {
    public void metodo() throws java.sql.SQLException {}
}
```