

# Strumenti e Linguaggi di Programmazione (SLP)

– Programmazione a oggetti –

Francesco Tiezzi



Scuola di Scienze e Tecnologie

Sezione di Informatica

Università di Camerino

- ▶ Programma: collezione di oggetti che interagiscono tra di loro per mezzo di azioni
  - ▶ Cambiando il proprio stato
  - ▶ Facendo cambiare lo stato degli altri oggetti
- ▶ Attenzione su dati da manipolare
  - ▶ Piuttosto che su procedure che eseguono manipolazione
- ▶ Progettazione di programma orientato a oggetti
  - ▶ Scomposizione di programma in tipi di dato (classi)
  - ▶ Definizione delle loro proprietà

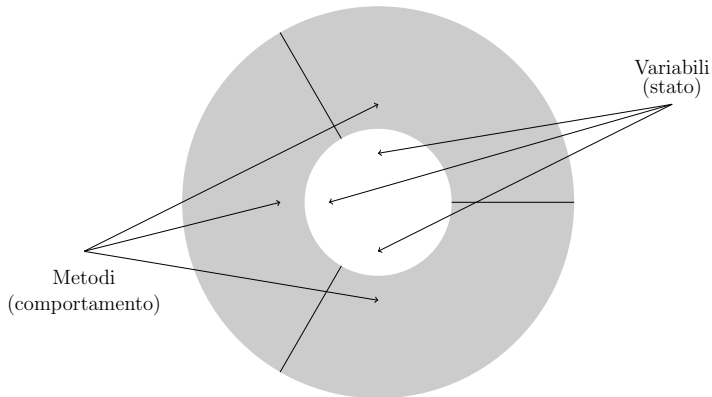
## Un esempio: la morra cinese

- ▶ Interazione tra due giocatori mediata da arbitro
  - ▶ Arbitro controlla legalità mosse effettuate
  - ▶ Dichiara vincitore di partita
- ▶ Ciascun “attore” corrisponde a **oggetto** software
  - ▶ Caratterizzato da *stato* e *comportamento*
    - ▶ Stato di giocatore: memoria dei turni di gioco
    - ▶ Comportamento: strategia adottata

## Strumenti e Linguaggi di Programmazione

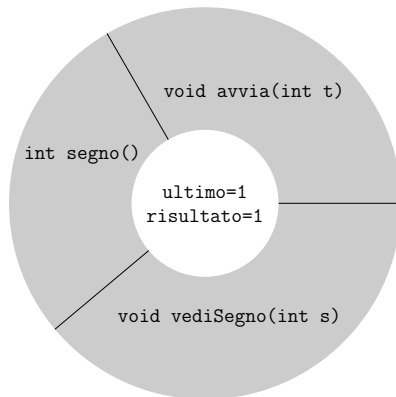
└ Introduzione alla programmazione a oggetti

└ Un esempio: la morra cinese



## Un esempio: la morra cinese

- ▶ Strategia “vinci-e-cambia, perdi-e-persisti”
  - ▶ Ripete stesso segno se ha perso e cambia se ha vinto
- ▶ Stato
  - ▶ Due variabili: ultimo e risultato
    - ▶ Sasso: 0
    - ▶ Forbici: 1
    - ▶ Carta: 2
    - ▶ Vittoria: 1
    - ▶ Sconfitta: 0



► Comportamento

► Inizio partita

```
void avvia( int t ) {  
    ultimo = 2;  
    risultato = 1;  
}
```

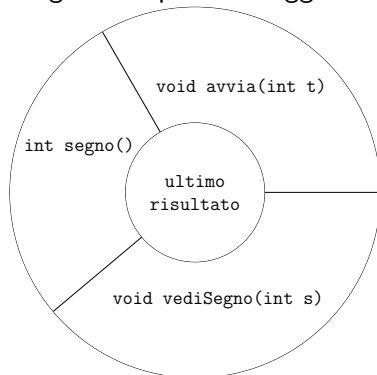
► Applica strategia

```
int segno() {  
    if (risultato == 1) {  
        ultimo = (ultimo + 1) % 3;  
    }  
    return ultimo;  
}
```

► "Vedi" segno avversario

```
void vediSegno(int s) {  
    if (s == (ultimo + 1) % 3) {  
        risultato = 1;  
    } else {  
        risultato = 0;  
    }  
}
```

- ▶ Necessari due giocatori
  - ▶ Se implementano stessa strategia: oggetti dello stesso tipo
    - ▶ Istanze della stessa **classe**
- ▶ Classe: fabbrica in grado di produrre oggetti dello stesso tipo



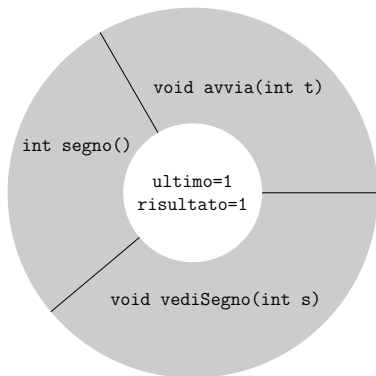
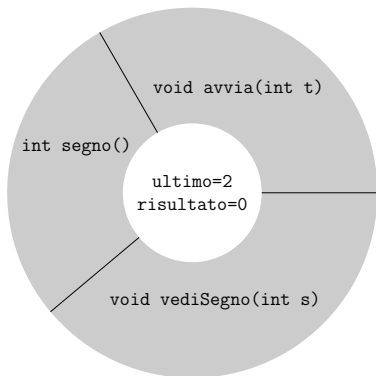
- ▶ Descrizione astratta di gruppo di oggetti con stesso insieme di variabili, che svolgono stesso insieme di operazioni in stesso modo



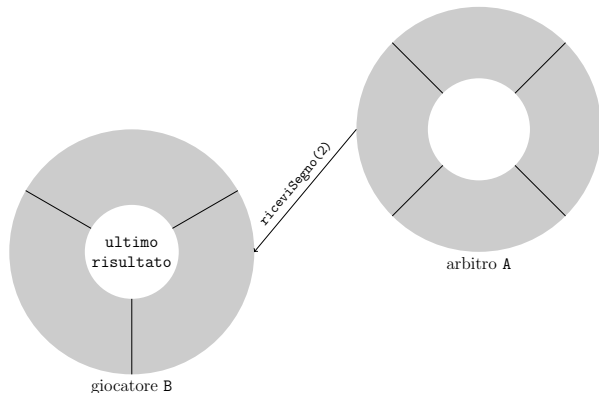
## ► Definizione di una classe

```
class VCPP {
    int ultimo;
    int risultato;
    void avvia( int t ) {
        ultimo = 2;
        risultato = 1;
    }
    int segno() {
        if (risultato == 1) {
            ultimo = (ultimo + 1) % 3;
        }
        return ultimo;
    }
    void vediSegno(int s) {
        if (s == (ultimo + 1) % 3) {
            risultato = 1;
        } else {
            risultato = 0;
        }
    }
}
```

- ▶ Una volta creata classe, possibile creare qualunque numero di oggetti della classe
  - ▶ Stato di oggetti cambia durante esecuzione programma



- ▶ Interazione tra giocatori attraverso interazione con arbitro
  - ▶ Interazione avviene tramite messaggi
    - ▶ **Messaggio**: invocazione di metodo di oggetto destinatario
    - ▶ Sintassi: nome oggetto seguito da "." e da invocazione



► Classe che modella arbitro

- Campo `nt`: numero di turni
- Metodo che esegue una partita
  1. comunica a giocatori inizio partita: invoca metodo `avvia` con argomento `nt`.
  2. Chiede a giocatori segno: invoca metodo `segno` e memorizza valore di ritorno
  3. Comunica a ciascun giocatore segno di avversario: invoca metodo `vediSegno` con argomento valore ricevuto da avversario
  4. Esamina segni, determina vincitore turno e aggiorna punteggi
  5. Se numero turni eseguiti minore di `nt`, esegue secondo passo. Altrimenti, determina vincitore

```
void gioca(VCPP g1, VCPP g2) {
    int[] p = { 0, 0 };
    g1.avvia(nt);
    g2.avvia(nt);
    int t = 0;
    while (t < nt) {
        t = t + 1;
        int s1 = g1.segno();
        int s2 = g2.segno();
        g1.vediSegno(s2);
        g2.vediSegno(s1);
        int v = vincitore(s1, s2);
        if (v > 0) {
            p[v - 1] = p[v - 1] + 1;
        }
    }
    if (p[0] == p[1]) {
        System.out.println("Nessun vincitore");
    } else {
        int r = p[0] > p[1] ? 1 : 2;
        System.out.println("Vince " + r);
    }
}
```



► Determinazione del vincitore turno

```
int vincitore(int segno1, int segno2) {  
    if (segno2 == (segno1 + 1) % 3) {  
        return 1;  
    }  
    if (segno1 == (segno2 + 1) % 3) {  
        return 2;  
    }  
    return 0;  
}
```

► Costruttore di arbitro

```
class Arbitro {
    int nt;
    Arbitro(int t) {
        nt = t;
    }
    void gioca(VCPP g1, VCPP g2) { ... }
    int vincitore(int segno1, int segno2) { ... }
}
```

- Usato in combinazione con operatore new per creare oggetti



► Classe principale

```
class Main {  
    public static void main(String[] args) {  
        Arbitro a = new Arbitro( 100 );  
        VCPP g1 = new VCPP();  
        VCPP g2 = new VCPP();  
        a.gioca(g1, g2);  
    }  
}
```

- Include metodo main
  - Sintassi specifica

## Creare e usare classi e oggetti

- ▶ Definizione di classe
  - ▶ **Dichiarazione**: prima linea di codice
    - ▶ Dichiarare almeno il nome
  - ▶ **Corpo**: segue dichiarazione
    - ▶ Racchiuso tra parentesi graffe
    - ▶ Contiene dichiarazione variabili d'istanza e metodi

```
class Rettangolo {  
    double larghezza;  
    double altezza;  
}
```

- ▶ Dichiarazione di oggetti appartenenti alla classe
  - ▶ Simile a dichiarazione di variabili di tipo primitivo  
`Rettangolo r;`
  - ▶ Crea **riferimento** a oggetto, non crea oggetto

## Creazione di oggetti

- ▶ Dichiarazione non crea oggetto

```
Rettangolo r;
```

- ▶ Riferimento uguale a null

- ▶ Creare oggetto: operatore new seguito da invocazione costruttore

- ▶ Sintassi

```
new nome_Classe( lista_Parametri )
```

- ▶ Esempio

```
Rettangolo r = new Rettangolo( );
```

```
Rettangolo r = new Rettangolo( 6, 10.5 );
```

## Distruzione di oggetti

- ▶ Non esistono distruttori espliciti
- ▶ *Garbage collector*: ricicla memoria non più referenziata
- ▶ Esempio

```
Rettangolo r = new Rettangolo( 6,10.5 );  
r = new Rettangolo( 7,12.3 );
```

- ▶ Memoria occupata da primo oggetto non più referenziata
- ▶ Garbage collector la rimette a disposizione per nuove allocazioni

## Usare gli oggetti

- ▶ Accesso a variabili e invocazione metodi

- ▶ All'interno della classe: semplicemente tramite nome
- ▶ All'esterno della classe: nome qualificato

- ▶ Sintassi

- riferimento\_Oggetto.nome\_Variabile*

- riferimento\_Oggetto.nome\_Metodo( lista\_Parametri )*

- ▶ Esempio

- `r.larghezza`

- `r.area( )`

## Parametri formali di tipo classe

- ▶ Simile a quanto visto con array
  - ▶ Oggetto può essere modificato da metodo
- ▶ Classe Rettangolo include metodo

```
void setLarghezza( double l ) {  
    larghezza = l;  
}
```

- ▶ Dall'esterno, possibile modificare larghezza rettangolo

```
void cambiaLarghezza( Rettangolo r, double l ) {  
    r.setLarghezza( l );  
}
```

- ▶ Sbagliato farlo in questo modo

```
void cambia( Rettangolo r, double l, double a ) {  
    Rettangolo nr = new Rettangolo( l, a );  
    r = nr;  
}
```

## I costruttori

- ▶ Sintassi

```
nome_Classe( lista_Parametri )  
blocco
```

- ▶ Esempio

```
 Rettangolo( double l, double a ) {  
     larghezza = l;  
     altezza = a;  
 }
```

- ▶ Servono a inizializzare le variabili d'istanza

- ▶ Spesso, costruttore senza parametri: inizializza con valori di default
- ▶ Nessun costruttore definito: Java ne fornisce uno di default senza parametri
  - ▶ Ma se un costruttore definito: quello di default non è fornito

## Il metodo main

- ▶ Metodo principale di una classe
  - ▶ Da esso Java inizia automaticamente esecuzione

- ▶ Sintassi

```
public static void main( String[] args )  
    blocco
```

- ▶ Esempio

```
public static void main( String[] args ) {  
    Rettangolo r = new Rettangolo( 3,4 );  
    System.out.println( r.getLarghezza( ) );  
}
```

- ▶ Se metodo main non esiste: errore segnalato da messaggio

```
java.lang.NoSuchMethodError: main
```



## Accessori e mutatori

- ▶ Consentono dall'esterno di accedere e modificare variabili di istanza
- ▶ Sintassi

```
tipo_Variabile getNome_Variabile( )
```

```
blocco
```

```
void setNome_Variabile( lista_Parametri )
```

```
blocco
```

- ▶ Esempio

```
double getLarghezza( ) {
```

```
    return larghezza;
```

```
}
```

```
void setLarghezza( double l ) {
```

```
    larghezza = l;
```

```
}
```

## La classe String

- ▶ Letterale di tipo String: stringa racchiusa tra doppi apici

```
String inizio = "Nuova partita";  
System.out.println( inizio );
```

- ▶ Costruttore con parametro una stringa

```
String inizio = new String( "Nuova partita" );
```

- ▶ Diverso da assegnazione diretta

```
String u = new String( "Nuova partita" );  
String v = new String( "Nuova partita" );  
String s = "Nuova partita";  
String t = "Nuova partita";  
System.out.println( u==v );  
System.out.println( s==t );
```

## Metodi della classe String

```
String stringa = "Nuova partita di morra";  
System.out.println( stringa.charAt( 0 ) );  
System.out.println( stringa.substring( 4 ) );  
System.out.println( stringa.substring( 2,10 ) );  
System.out.println( stringa.indexOf( "a" ) );  
System.out.println( stringa.indexOf( "a",5 ) );  
System.out.println( stringa.lastIndexOf( "a" ) );
```

▶ Operatore di concatenazione: +

- ▶ Possibile concatenare numero qualsiasi di oggetti di tipo String

```
String primo = "Buongiorno!";  
String secondo = "Mi chiamo Pierluigi.";  
messaggio = primo+" "+secondo;
```

- ▶ Possibile concatenare oggetto di tipo String con altro tipo di oggetto

```
String altezza = "L'altezza e' "+5.1;
```

▶ Sequenze escape: consentono di includere caratteri speciali

```
System.out.println( "L'altezza e' \"5.1\"." );
```

- ▶ Altri esempi: `\\` e `\n`
- ▶ String: solo una di tante classi a disposizione in librerie Java
  - ▶ Consultare documentazione Javadoc

## Sovraccaricamento dei metodi

- ▶ **Firma** di un metodo

  - ▶ Nome

  - ▶ Numero e tipo dei parametri

- ▶ Metodi con firma diversa per il nome

```
int mioMetodo( int i )
```

```
int tuoMetodo( int i )
```

- ▶ Metodi con firma diversa per il numero di parametri

```
int mioMetodo( int i )
```

```
int mioMetodo( int i, double j )
```

- ▶ Metodi con firma diversa per il tipo di parametri

```
int mioMetodo( int i, double j )
```

```
int mioMetodo( double j, int i )
```

- ▶ Metodi con la stessa firma

```
int mioMetodo( int i )
```

```
double mioMetodo( int j )
```

## Sovraccaricamento dei metodi

- ▶ In una classe, metodi devono avere firma diversa
  - ▶ Metodi con stesso nome: **sovraccaricati**
    - ▶ Tipico esempio: costruttori
- ▶ Esempio

```
class MetodiSovraccaricati {  
    int square(int x) {  
        return x*x;  
    }  
    double square(double y) {  
        return y*y;  
    }  
    int square(double x) {  
        return x*x;  
    }  
}
```

## Metodi sovraccaricati

- ▶ Compilatore determina quale definizione usare
  - ▶ In base a numero e tipo di argomenti
  - ▶ Eventualmente, effettuando qualche conversione di tipo
- ▶ Se non ci riesce, invia messaggio di errore
- ▶ Esempio

```
int mioMetodo( int i, double j )
```

```
int mioMetodo( double j, int i )
```

Invocazione `mioMetodo( 5,10 )` genera errore di compilazione

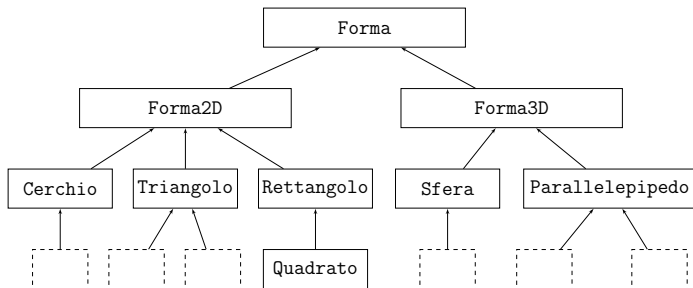
## Ereditarietà

- ▶ Concetto chiave della programmazione a oggetti
  - ▶ Permette di usare classe esistente, per definire nuova classe
    - ▶ Facilita riutilizzo del codice
    - ▶ Semplifica stesura, correzione e mantenimento di correttezza
- ▶ Definizioni
  - ▶ *Classe padre* (*classe base* o **super-classe**)
    - ▶ Classe molto generale
  - ▶ *Classe figlia* (*classe derivata* o **sotto-classe**)
    - ▶ Aggiunge nuovi dettagli a definizione classe generale
    - ▶ Può utilizzare variabili di istanza e metodi di classe padre
    - ▶ Può utilizzare campi e metodi aggiunti
    - ▶ Può ridefinire metodi di classe padre, modificandone comportamento



## Ereditarietà

- Gerarchia delle classi
  - Indica relazioni di ereditarietà



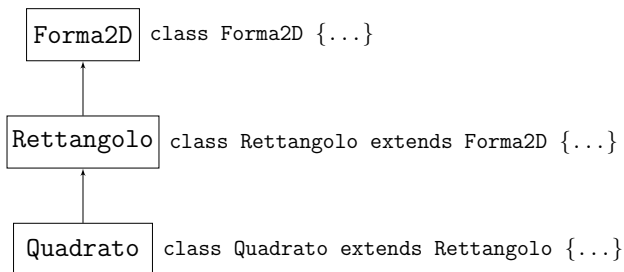
## Ereditarietà

### ► Sintassi

```
class nome_Sotto-classe extends nome_Super-classe  
blocco
```

### ► Esempio

```
class Quadrato extends Rettangolo {  
    // corpo della classe  
}
```



## Costruttori ed ereditarietà

- ▶ Costruttore di classe derivata
  - ▶ Per prima cosa invoca costruttore classe padre
    - ▶ Usa parola chiave `super` con parametri appropriati
  - ▶ Se invocazione non inclusa, Java include automaticamente invocazione costruttore di default (senza parametri)
    - ▶ Se costruttore di default non presente in super-classe: errore di compilazione

## ▶ Esempio corretto

```
class Forma2D {
    double larghezza;
    double altezza;
    Forma2D(double l, double a) {
        larghezza = l;
        altezza = a;
    }
}

class Rettangolo extends Forma2D {
    Rettangolo() {
        super( 0,0 );
    }
    Rettangolo(double l, double a) {
        super( l,a );
    }
}
```

## ▶ Esempio sbagliato

```
class Forma2D {
    double larghezza;
    double altezza;
    Forma2D(double l, double a) {
        larghezza = l;
        altezza = a;
    }
}

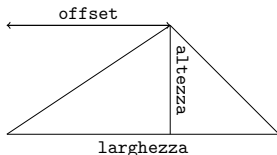
class ErroreRettangolo1 extends Forma2D {
    ErroreRettangolo1( double l,double a ) {
        System.out.println( "Costruzione rettangolo" );
        super( l,a );
    }
}

class ErroreRettangolo2 extends Forma2D {
    ErroreRettangolo2( ) {
        super( );
    }
}

class ErroreRettangolo3 extends Forma2D {
}
```

## Variabili di istanza, metodi ed ereditarietà

- ▶ Classe figlia eredita variabili d'istanza di classe padre
- ▶ Classe figlia può aggiungere ulteriori variabili d'istanza
- ▶ Esempio



```
class Triangolo extends Forma2D {  
    int offset;  
    Triangolo(double l,double a,double o) {  
        super( l,a );  
        offset = o;  
    }  
}
```

- ▶ Classe derivata può aggiungere nuovi metodi non inclusi in classe padre
- ▶ Esempio

```
class Rettangolo extends Forma2D {  
    Rettangolo() {  
        super( 0,0 );  
    }  
    Rettangolo( double l,double a ) {  
        super( l,a );  
    }  
    double area() {  
        return larghezza*altezza;  
    }  
    double perimetro() {  
        return 2*(larghezza+altezza);  
    }  
}
```

- ▶ Classe derivata può riscrivere metodi di classe padre
- ▶ Esempio

```
class Quadrato extends Rettangolo {  
    Quadrato() {  
    }  
    Quadrato(double l) {  
        super( l,l );  
    }  
    double perimetro() {  
        return 4*larghezza;  
    }  
}
```

- ▶ Per invocare metodo di classe padre: usare parola chiave `super` seguita da punto e nome metodo sovrascritto
- ▶ Non confondere sovrascrittura con sovraccaricamento



## Tipi di dato ed ereditarietà

- ▶ Conseguenza di ereditarietà: oggetto può avere più di un tipo
  - ▶ Tipo classe derivata
  - ▶ Tipo classe padre
  - ▶ Se classe padre è derivata, numero di tipi aumenta
- ▶ Oggetto di classe derivata usabile quando è ammesso usare oggetto di classe padre
  - ▶ Analogo a tipi primitivi
- ▶ Esempio
  - ▶ Metodo di classe A con parametro formale di tipo classe B
  - ▶ Argomento può essere oggetto di classe C derivata da B
- ▶ Catena di ereditarietà include classe antenata di tutte le classi: classe Object

- ▶ Possibile assegnare a variabile di classe padre oggetto di classe derivata

- ▶ Analogo a tipi primitivi
- ▶ Esempio

```
 Rettangolo r;  
 Quadrato q = new Quadrato( 3 );  
 r = q;
```

- ▶ Non possibile il contrario

- ▶ Analogo a tipi primitivi
- ▶ Esempio

```
 Rettangolo r = new Rettangolo( 3,2 );  
 Quadrato q;  
 q = r;
```

## Classi astratte

- ▶ Una **classe astratta** è una classe dichiarata `abstract`
  - ▶ Una classe astratta può includere metodi astratti
  - ▶ Una classe astratta **NON** può essere istanziata
  - ▶ Una classe astratta può essere estesa
- ▶ Un **metodo astratto** è un metodo dichiarato senza implementazione, cioè senza parentesi graffe e seguito da ;  

```
abstract int mioMetodo( int i, double j );
```
- ▶ Se una classe include metodi astratti, allora la classe stessa deve essere astratta
- ▶ Se una classe astratta viene estesa, la classe derivata tipicamente fornisce implementazioni di tutti i metodi astratti; se non lo fa, anche la classe derivata deve essere astratta

## Classi astratte

- ▶ Esempio di classe astratta:

```
abstract class Forma2D {
    double larghezza;
    double altezza;

    Forma2D(double l, double a) {
        larghezza = l;
        altezza = a;
    }

    abstract double area();
    abstract double perimetro();
}
```

- ▶ Il costruttore Forma2D può essere usato dalle classi derivate

## Interfacce

- ▶ Un'interfaccia è una sorta di *contratto* che definisce come diverse classi dovranno interagire
- ▶ Tecnicamente, un'interfaccia è un tipo, simile ad una classe, che può contenere solo:
  - ▶ firme di metodi e costanti
- ▶ Un'interfaccia NON può contenere dettagli implementativi:
  - ▶ metodi implementati e variabili di istanza

- ▶ Sintassi interfaccia:

```
interface nome_interfaccia  
blocco
```

- ▶ Sintassi implementazione interfaccia:

```
class nome_classe implements nome_interfaccia  
blocco
```

## Interfacce: esempio

```
interface Forma2D_Interfaccia {
    // Constanti: es. PI_GRECO = 3.14

    double area ();
    double perimetro ();
}

class Rettangolo implements Forma2D_Interfaccia {
    double larghezza;
    double altezza;

    Rettangolo(double l, double a)
        {larghezza = l; altezza = a;}

    double area(){return larghezza*altezza;}
    double perimetro(){return 2*(larghezza+altezza);}
}
```

## Modificatori

- Influiscono su modo con cui si può accedere e su comportamento di componente modificata

	<b>campo</b>	<b>metodo</b>	<b>classe</b>
<code>public</code>	✓	✓	✓
<code>private</code>	✓	✓	
<code>static</code>	✓	✓	
<code>final</code>	✓	✓	✓

## Modificatori di accesso

- ▶ `public`: nessuna restrizione su chi può utilizzare classi, variabili o metodi

```
public class A {
    public int x;
    public int getX() {
        return x;
    }
}

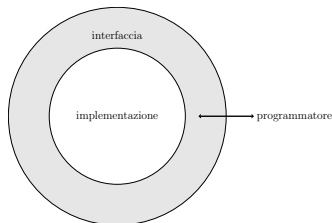
class B {
    public static void main(String[] args) {
        A a = new A( );
        a.x = 3;
        System.out.println( a.getX( ) );
    }
}
```



- ▶ `private`: non accessibili fuori dalla classe

```
class CampiMetodiPrivati {
    private int campo;
    CampiMetodiPrivati(int inCampo) {
        campo = inCampo;
    }
    private int valoreDefault() { return 100; }
    public void setCampoDefault() {
        campo = valoreDefault( );
    }
    public int getCampo() { return campo; }
}
class TestCampiMetodiPrivati {
    public static void main( String[] args ) {
        CampiMetodiPrivati s = new CampiMetodiPrivati(10);
        s.campo = s.valoreDefault( );
        s.setCampoDefault( );
        System.out.println( s.getCampo( ) );
    }
}
```

## ► Incapsulamento



- Commento prima di definizione classe
- Variabili di istanza `private`
- Metodi accessori e mutatori `public`
- Metodi `public` per altre azioni basilari
- Commento prima di intestazione metodi pubblici
- Privato qualsiasi metodo di supporto a quelli pubblici

## Modificatori di comportamento

- ▶ `static`: metodi e variabili appartengono a tutta la classe e non richiedono oggetto per essere utilizzati
  - ▶ Per questo, metodo `main` statico e pubblico
- ▶ Si usa nome di classe (oltre a nome di oggetto)
  - ▶ Esempio
    - ▶ `System.out`
    - ▶ `Math.random()`

## Il gioco dell'oca

- ▶ Classe Oca
  - ▶ Due variabili di istanza
    - ▶ Tabellone: array unidimensionale con penalità e premi
    - ▶ Array delle posizioni dei giocatori

```
class Oca {  
    int[] tabellone = { 0, 0, 0, 0, 0, 0, 0, 0, 0,  
        0, 20, 0, 0, 0, 0, 0, 0, 0,  
        0, 0, 0, 33, 0, 0, 0, 0, 0,  
        0, 35, 0, 0, 0, 0, 19, 0, 0,  
        0, 48, 0, 0, 15, 0, 0, 0, 0,  
        0, 0, 0, 0, 0, 40, 0, 0, 0,  
        0, 0, 0, 8, 0, 0 };  
    int[] posizione;  
    ...  
}
```

▶ Classe Oca

- ▶ Costruttore pubblico con parametro il numero dei giocatori
  - ▶ Sfasamento di indici

```
public Oca(int ng) {  
    posizione = new int[ng];  
    for (int i = 0; i < ng; i = i+1) {  
        posizione[i] = 1;  
    }  
}
```

► Classe Oca

- Metodo privato run che simula una partita

```
private void run() {  
    int ng = posizione.length;  
    int turno = -1;  
    boolean finito = false;  
    while (!finito) {  
        turno = (turno+1)%ng;  
        finito = muoviGiocatore( turno );  
    }  
    System.out.println( "Vince "+(turno+1) );  
}
```

## ► Classe Oca

- Metodo privato `muoviGiocatore` per muovere giocatore e verificare se ha vinto
  - Usa metodo statico `random` della classe `Math`

```
private boolean muoviGiocatore(int t) {
    int dado = 1+(int) (Math.random( )*6);
    posizione[t] += dado;
    if (posizione[t]>tabellone.length) {
        posizione[t] = 2*tabellone.length-posizione[t];
    }
    if (tabellone[posizione[t]-1]>0) {
        posizione[t] = tabellone[posizione[t]-1];
    }
    return (posizione[t]==tabellone.length);
}
```

- ▶ Classe Oca

- ▶ Metodo main

- ▶ Usa argomenti da linea di comando

```
public static void main(String[] args) {  
    if (args.length!=1) {  
        System.out.println( "Uso: java Oca <ng>" );  
        System.exit( 0 );  
    }  
    int ng = (new Integer( args[0] )).intValue( );  
    Oca o = new Oca( ng );  
    o.run();  
}
```



## Final

- ▶ `final`
  - ▶ Variabili: non possono essere modificate (in pratica, costanti)
  - ▶ Metodi: non possono essere sovrascritti
    - ▶ Metodo `private`: metodo finale
  - ▶ Classi: non possono essere usate come super-classe
    - ▶ Tutti i metodi implicitamente finali
    - ▶ Esempio: classe `System`

## Eccezioni

- ▶ Consentono di gestire situazioni anomale che si possono verificare durante l'esecuzione
- ▶ Quando Java VM si trova in una situazione anomala
  - ▶ Sospende il programma
  - ▶ Crea un oggetto della classe corrispondente all'anomalia
  - ▶ Passa il controllo a un gestore di eccezioni (implementato dal programmatore)
    - ▶ Se il programmatore non ha previsto nessun gestore, interrompe il programma e stampa il messaggio di errore
- ▶ Programmatore gestisce anomalia tramite costrutto `try-catch`
  - ▶ Monitora porzione di programma
  - ▶ Specifica cosa fare in caso si verifichi anomalia in porzione di programma monitorata

## La gerarchia delle eccezioni

- ▶ La classe `Exception` descrive eccezione generica
  - ▶ Situazioni anomale più specifiche: descritte da sottoclassi di `Exception`
- ▶ Costrutto `try-catch` può gestire più tipi di eccezione contemporaneamente
  - ▶ Vari gestori (ovvero `catch`) controllati in sequenza
  - ▶ Eseguito (solo) il primo `catch` che prevede un tipo di eccezione che è superclasse dell'eccezione che si è verificata
    - ▶ Meglio non mettere `Exception` per prima
- ▶ Eccezioni si dividono in
  - ▶ `Checked`: il compilatore richiede che ci sia un gestore
    - ▶ Esempio: `FileNotFoundException`
    - ▶ Quelle definite dal programmatore
  - ▶ `Unchecked`: il gestore non è obbligatorio
    - ▶ Esempio: `ArrayIndexOutOfBoundsException`
    - ▶ Sottoclasse di `RuntimeException`

## Lanciare eccezioni

- ▶ Comando `throw`: consente di lanciare un'eccezione quando si vuole
  - ▶ Si può usare classe `Exception`, sua sottoclasse già definita, o sua sottoclasse definita da programmatore
  - ▶ `throw` seguito da un oggetto, solitamente costruito al momento (tramite `new`)
  - ▶ Costruttore di eccezione può ricevere parametri
    - ▶ Esempio: stringa di descrizione
- ▶ Utilizzo di `throw` dentro a un metodo: interrompe il metodo in caso di situazioni anomale
  - ▶ Chi invoca il metodo dovrà preoccuparsi di implementare un gestore delle eccezioni possibilmente sollevate
  - ▶ Evitare valori di ritorno dei metodi che servono solo a dire se l'operazione è andata a buon fine
- ▶ Metodo che contiene dei comandi `throw` deve elencare eccezioni sollevate
  - ▶ Parola chiave `throws`

## Pacchetti

- ▶ Meccanismo che consente di raggruppare le classi
  - ▶ Libreria Standard di Java organizzata in pacchetti
    - ▶ `String`: classe del pacchetto `java.lang`
    - ▶ `FileReader`: classe del pacchetto `java.io`
- ▶ Pacchetto riunisce classi logicamente correlate tra loro
  - ▶ `java.lang`: classi fondamentali del linguaggio Java (come `String`)
  - ▶ `java.util`: classi di frequente utilizzo (come `Random`)
  - ▶ `java.awt` e `java.swing`: classi per costruire interfacce grafiche
- ▶ Programma complesso: raggruppamento in pacchetti consente di fare ordine
  - ▶ Nel caso del `MasterMind`
    - ▶ `core`: classi che costituiscono nucleo del programma
    - ▶ `gui`: classi che realizzano interfaccia grafica
    - ▶ `player`: classi di giocatori
    - ▶ `exception`: classi delle possibili eccezioni

## Definizione e struttura dei pacchetti

- ▶ Comando `package` all'inizio del file Java
  - ▶ Esempio: `package core;`
- ▶ File java delle varie classi salvati in diverse directory che corrispondono ai vari pacchetti
  - ▶ Compilatore produce errore se i file non sono nelle directory giuste
  - ▶ Nessun pacchetto: default corrispondente alla directory principale
- ▶ Pacchetti possono essere ulteriormente raggruppati, formando struttura gerarchica
  - ▶ Raggruppamento tramite prefisso nel nome del pacchetto
    - ▶ Nel caso del MasterMind: prefisso `it.unifi.mastermind`
  - ▶ Prefisso si riflette in struttura delle directory

## Pacchetti e visibilità

- ▶ `private`: utilizzabile solo all'interno della stessa classe
- ▶ senza modificatore: utilizzabile solo nel pacchetto che contiene la classe
- ▶ `protected`: utilizzabile nel pacchetto che contiene la classe, e in tutte le classi che ereditano da essa
- ▶ `public`: utilizzabile ovunque

## File di testo: lettura

- ▶ File contenenti caratteri
  - ▶ Esistono anche file binari
- ▶ Loro gestione fa uso di classi in pacchetto `java.io`
- ▶ Modo più semplice
  - ▶ Creare un oggetto di tipo `FileReader`
    - ▶ Richiede il nome del file nel costruttore
    - ▶ `FileReader fr = new FileReader("config.txt");`
    - ▶ Può generare `FileNotFoundException`
  - ▶ Collegare al lettore di file un `BufferedReader`
    - ▶ Richiede il lettore di file nel costruttore
    - ▶ `BufferedReader in = new BufferedReader(fr);`
    - ▶ Fornisce metodo `readLine` per leggere una riga alla volta
    - ▶ `String line = in.readLine();`
    - ▶ Può generare `IOException`
  - ▶ Chiudere il lettore alla fine
    - ▶ `in.close();`