

Corso 1 Java

Programmazione Base

Giuseppe Dell'Abate

A series of horizontal lines of varying lengths and colors (teal, light blue, and white) extending from the right side of the slide.

Modulo 1

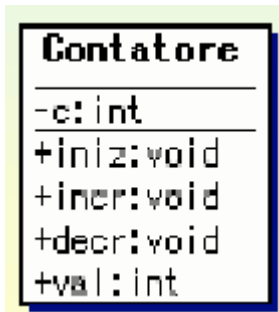
Concetti della programmazione orientata agli oggetti

A series of horizontal lines in teal and light blue colors, of varying lengths, extending from the left edge of the slide towards the right, positioned below the main title.

Lezione 1

Classi

- I più diffusi linguaggi ad oggetti sono basati sul concetto di classe come insieme di oggetti con struttura e comportamento simili
- La classe definisce un tipo
- Istanza di una classe = oggetto della classe



Lezione 1

Classi

- Una classe realizza l'implementazione di un tipo di dato astratto.
- In Java:

```
class Contatore {  
    int c;  
    void iniz(int i) {c = i;}  
    void incr() {++c;}  
    void decr() {--c;}  
    int val() {return c;}  
}
```

Lezione 2

Ereditarietà

- L'ereditarietà consente di definire un tipo (=classe) sulla base dei dati e dei metodi di un tipo già definito, ereditandone operativamente sia i dati che i metodi.
- La classe di partenza è la "classe base" o "superclasse" o "classe padre"; mentre la classe che si crea è la "classe derivata" o la "sottoclasse" o la "classe figlia".
- Nella terminologia OOP si usa anche il termine "estendere" che indica la creazione di una classe derivata da una classe base.
- Non bisogna mai confondere l'ereditarietà con l'incorpamento (classe contenitore)!
- Di grosso aiuto sono, nel linguaggio corrente, i verbi "E' un" e "Ha un".

Lezione 2

Ereditarietà

- Esempio:

```
class Padre {  
    int x = 5;  
}  
public class Figlio extends Padre {  
    public static void main(String[] args) {  
        Figlio pr = new Figlio();  
        pr.leggo();  
    }  
    void leggi() {  
        System.out.println(x);  
    }  
}
```

- La classe Figlio eredita la proprietà x dalla classe Padre e può usarla al suo interno

Lezione 3

Polimorfismo

- Derivato dal greco, significa “pluralità di forme”
- E’ la caratteristica che ci consente di utilizzare un’unica interfaccia per una moltitudine di azioni. Quale sia la particolare azione eseguita dipende solamente dalla situazione in cui ci si trova.
- Mentre con gli oggetti il collegamento con la classe avviene a compile-time, nel caso del polimorfismo avviene a run-time
- Il polimorfismo viene impiegato impiegando una interfaccia, una classe astratta o una classe.
- E’ necessario che ci sia ereditarietà tra le classi

Lezione 3

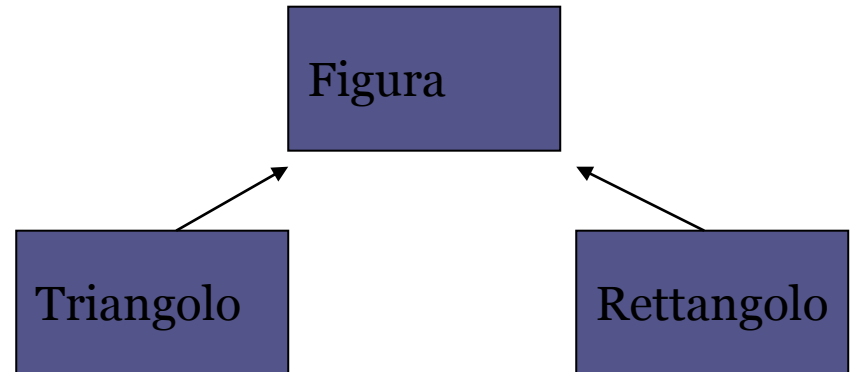
Polimorfismo

- Esempio:

```
abstract class Figura {  
    abstract void rispondo();  
}
```

```
class Triangolo extends Figura {  
    void rispondo(){System.out.println("Sono il triangolo");}  
}
```

```
class Rettangolo extends Figura {  
    void rispondo(){System.out.println("Sono il  
    rettangolo");}  
}
```



Lezione 3

Polimorfismo

- Continua... esempio:

```
public class Altro {  
    Figura pr;  
    public static void main(String[] args) {  
        Altro al = new Altro();  
        al.metodo();  
    }  
    void metodo() {  
        pr = new Triangolo();  
        pr.rispondo();  
        pr = new Rettangolo();  
        pr.rispondo();  
    }  
}
```

- Tutto ciò è possibile se esiste l'ereditarietà!

Lezione 4

Astrazioni

- Le astrazioni non dovrebbero dipendere dai dettagli. I dettagli dovrebbero dipendere dalle astrazioni
- Le astrazioni contengono pochissimo codice (in teoria nulla) e quindi sono poco soggette a cambiamenti.
- I moduli non astratti sono soggetti a cambiamenti ma questi cambiamenti sono sicuri perché nessuno dipende da questi moduli
- I moduli più dettagliati (concreti) dipendono da moduli meno dettagliati (astratti).

Lezione 4

Astrazioni

- Abbiamo colto i due obiettivi fondamentali:
 1. I dettagli del sistema sono stati isolati fra di loro, separati da un muro di astrazioni stabili, e questo impedisce ai cambiamenti di propagarsi. (design for change)
 2. Nel contempo i singoli moduli sono maggiormente riusabili perché sono disaccoppiati fra di loro (design for reuse).

Modulo 2

Introduzione a java

A series of horizontal lines in teal and light blue colors, of varying lengths, extending from the left edge of the slide to the right, positioned below the title.

Lezione 1

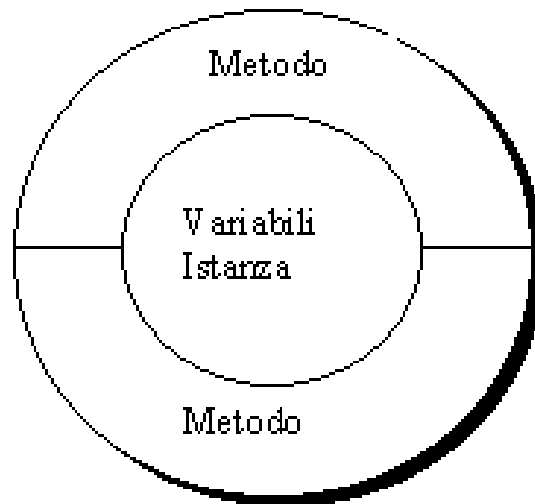
Caratteristiche del linguaggio

- orientato agli oggetti
- semplice
- indipendente dalla piattaforma
- portabile
- sicuro
- multithreading
- dinamico

Lezione 1

Orientato agli oggetti

- Consente la creazione di TIPI personalizzati
- Basato sulle classi NON sugli oggetti



Lezione 1

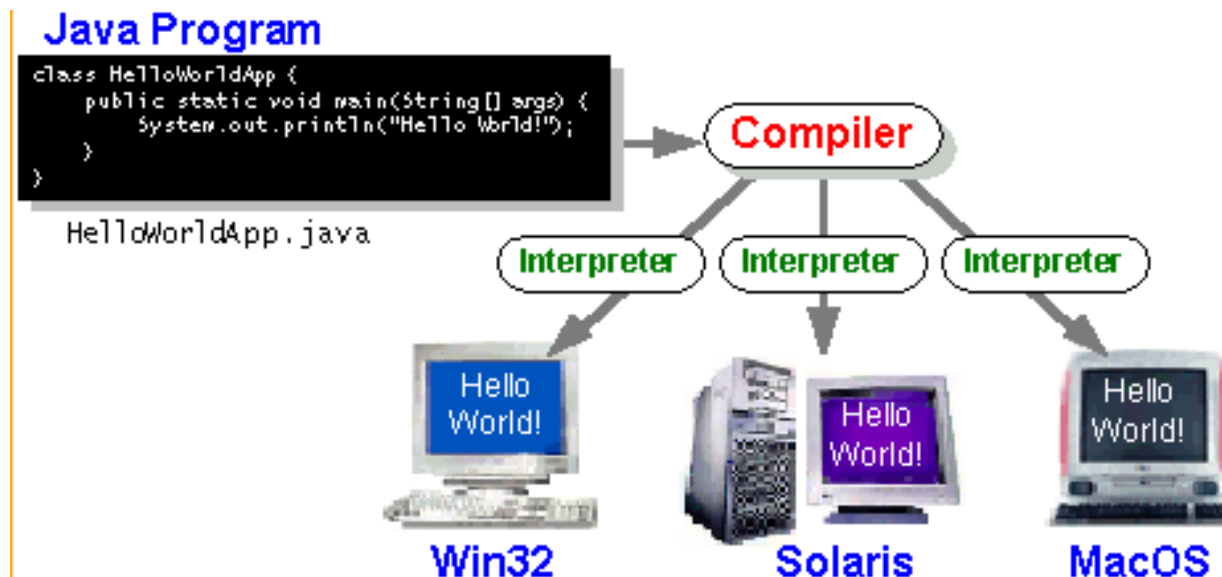
Semplice

- Rispetto al C++
 1. non è object based ma object oriented (ad oggetti puro)
 2. non ha l'aritmetica dei puntatori (passaggi per valore)
 3. non deve deallocare la memoria (garbage collector)
 4. non ha ereditarietà multipla (singola)

Lezione 1

Indipendente dalla piattaforma

- Indipendentemente dalla architettura hardware e dal sistema operativo, la Java Virtual Machine interpreta il bytecode e lo esegue



Lezione 1

Portabile

- Tipi primitivi mantengono le loro dimensioni in tutte le piattaforme grazie alla JVM

<i>Primitiva</i>	<i>Dimensione</i>
boolean	1-bit
char	16-bit
byte	8-bit
short	16-bit
int	32-bit
long	64-bit
float	32-bit
double	64-bit

Lezione 1

Dinamico

- Le classi sono collegate solo quando richiesto
- Possono provenire dal file system locale così come dalla rete
- Il codice viene verificato prima di essere passato all'interprete per l'esecuzione.
- E' possibile aggiungere nuovi metodi e variabili di istanza ad una classe, senza dover ricompilare l'intera applicazione

Lezione 1

MultiThreading

- Elaborazioni simultanee in un ambiente a singolo processo
- La gestione dei thread in genere è di tipo preemptive
- Nei sistemi in cui la gestione è non preemptive Java fornisce il metodo `yield()` che dà, ad un altro thread, la possibilità di essere comunque eseguito
- I metodi dichiarati `synchronized` non possono essere eseguiti simultaneamente da più thread;

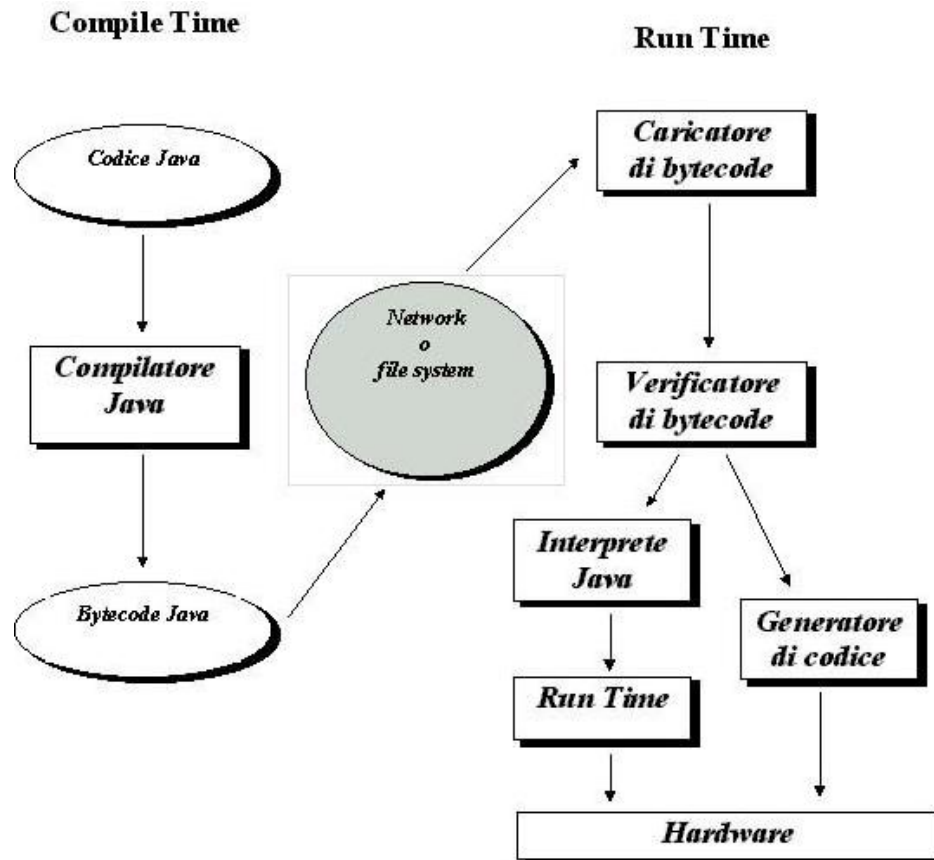
Lezione 1

Sicuro

- Operazioni eseguite dal compilatore:
 1. controlli di errori di codifica
 2. le dichiarazioni di tipo esplicite e non implicite
 3. le classi sono inserite in un name-space per gestire la sicurezza
- Operazioni eseguite dal sistema run-time:
 1. allocazione della memoria dipende dalla piattaforma
 2. puntatore alla memoria non esiste ma riferimenti simbolici, risolti in indirizzi di memoria reali in fase di esecuzione
 3. I tipi sono noti e corretti
- Operazioni eseguite dal compilatore e dal sistema run-time :
 1. strati di difesa contro il codice che può essere potenzialmente pericoloso

Lezione 2

La JVM, portabilità e riusabilità



Lezione 3

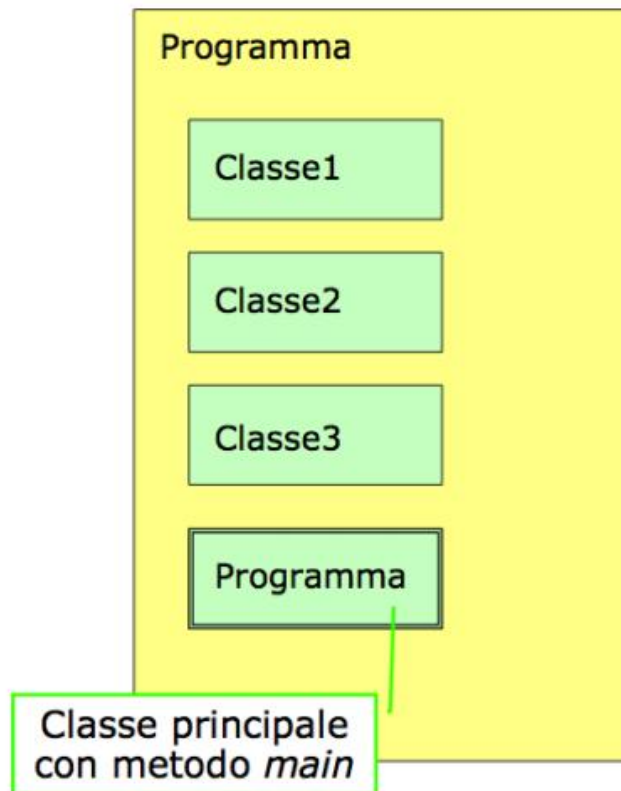
Il metodo main, compilazione ed esecuzione

Primo programma in JAVA

- Un programma Java è costituito da una o più classi
- Ogni classe risiede in un file che ha lo stesso nome della classe ed estensione .java
- Deve esistere almeno una classe che ha il nome del programma ed un metodo speciale chiamato main.

Lezione 3

Il metodo main, compilazione ed esecuzione



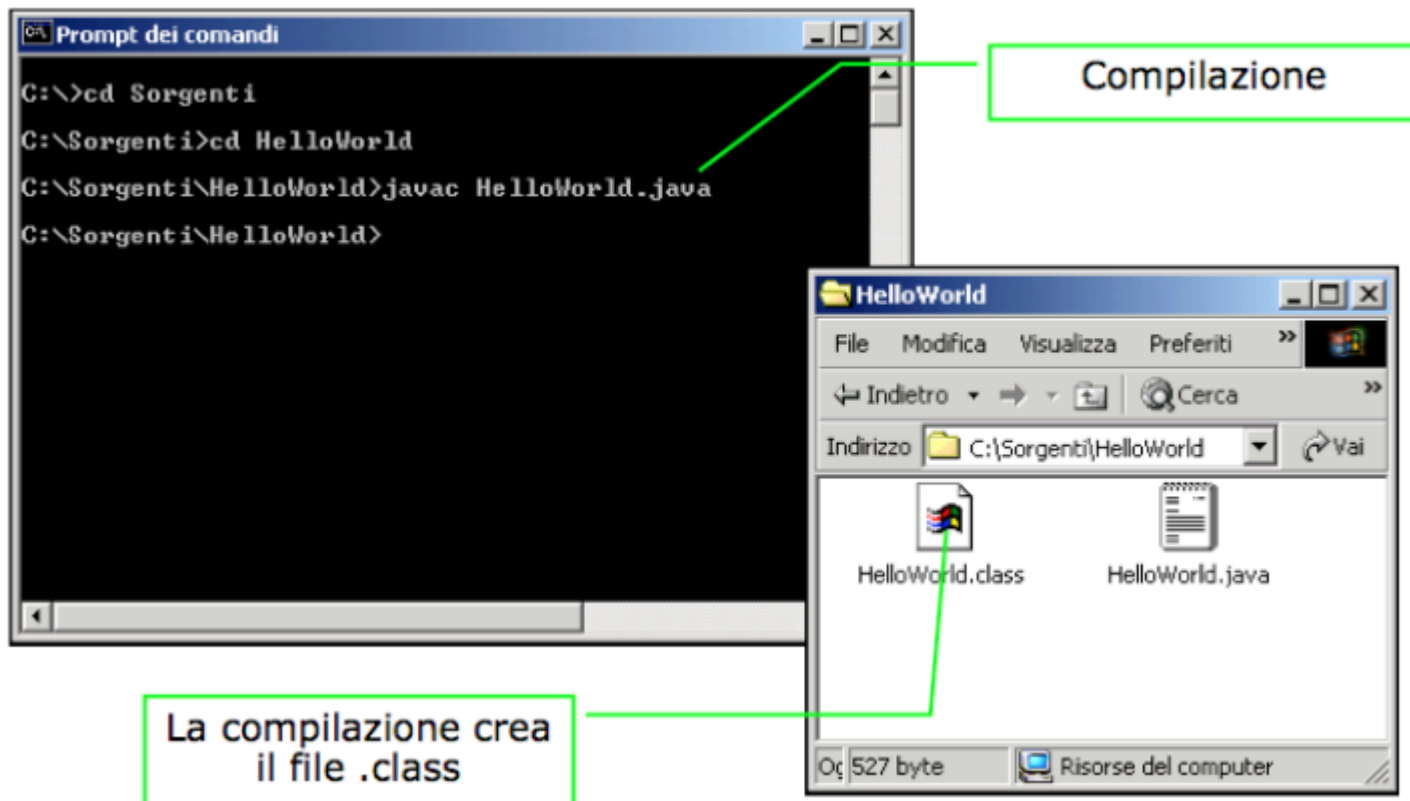
Lezione 3

Il metodo main, compilazione ed esecuzione

```
public class HelloWorld{  
    public static void main (String args[] ){  
        System.out.println("Hello");  
    }  
}
```


Lezione 3

Il metodo main, compilazione ed esecuzione



Lezione 3

Il metodo main, compilazione ed esecuzione



```
Prompt dei comandi

C:\Sorgenti\HelloWorld>java HelloWorld
Hello

C:\Sorgenti\HelloWorld>
```

The image shows a screenshot of a Windows Command Prompt window. The title bar reads "Prompt dei comandi". The command prompt shows the directory "C:\Sorgenti\HelloWorld". The user has entered the command "java HelloWorld", and the output is "Hello". The prompt is now waiting for the next command.

Lezione 3

Il metodo main, compilazione ed esecuzione

- Per compilare ed eseguire la classe HelloWorld

```
C:\Sorgenti\HelloWorld\javac HelloWorld.java
```

```
C:\Sorgenti\HelloWorld \java HelloWorld
```

```
Hello
```

```
C:\Sorgenti\HelloWorld
```

Lezione 3

Il metodo main, compilazione ed esecuzione

- Il metodo main è il metodo della classe che viene eseguita per avviare l'applicazione.
- Esiste un solo metodo main
- Il metodo main viene usato solo per le applicazioni e non per le Applet o le servlet.

```
public static void main(String [] args) {  
    //blocco di istruzioni  
}
```

Lezione 3

Il metodo main, compilazione ed esecuzione

1. **public**: è necessario che sia accessibile per poter avviare la classe
2. **static**: è un metodo di classe
3. **void**: non restituisce nessun valore
4. **main**: nome del metodo principale,
5. **String [] args**: Array di stringhe usato come parametro (obbligatorio)
6. **blocco di istruzioni**: sono le istruzioni che implementiamo

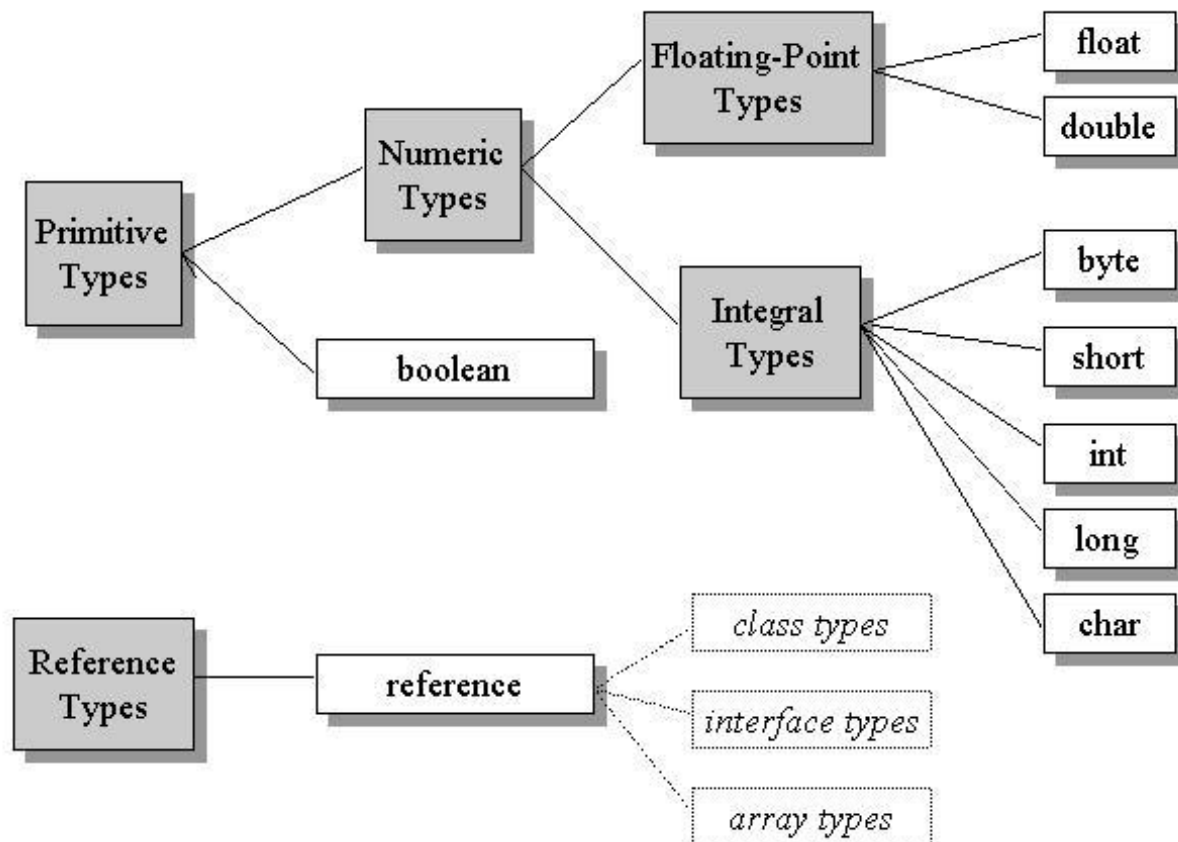
Lezione 4

Tipi primitivi e costrutti di controllo

- Classe Java è composta da:
 1. **metodi speciali** (costruttore, main)
 2. **metodi** (funzioni, procedure)
 3. **proprietà** (variabile, costante)
- Le proprietà sono collegate ai tipi, che posso essere :
 1. **primitivi**: si prestano ad un uso intuitivo
 2. **reference**: fanno riferimento ad oggetti, array, ecc

Lezione 4

Tipi primitivi e costrutti di controllo



Lezione 4

Tipi primitivi e costrutti di controllo

- Per garantire la portabilità del Bytecode da una piattaforma ad un'altra Java fissa le dimensioni di ogni dato primitivo

<i>Primitiva</i>	<i>Dimensione</i>	<i>Val. minimo</i>	<i>Val. Massimo</i>
boolean	1-bit	-	-
char	16-bit	Unicode 0	Unicode $2^{16} - 1$
byte	8-bit	-128	+127
short	16-bit	-2^{15}	$+2^{15} - 1$
int	32-bit	-2^{31}	$+2^{31} - 1$
long	64-bit	-2^{63}	$+2^{63} - 1$
float	32-bit	IEEE754	IEEE754
double	64-bit	IEEE754	IEEE754
void	-	-	-

Lezione 4

Tipi primitivi e costrutti di controllo

- Java assegna ad ogni variabile un valore di default al momento della dichiarazione

Tipo primitivo	Valore assegnato dalla JVM
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0

Lezione 4

Numeri interi

- Il tipo **int** è senza dubbio il tipo primitivo più usato, per ragioni di praticità e di efficienza. Può contenere qualunque numero intero compreso tra 2.147.483.647 e -2.147.483.648, ossia tutti i numeri rappresentabili con una cella di memoria a 32 bit
- Il tipo **byte** permette di operare su numeri compresi tra -128 e 127, che sono i numeri che è possibile rappresentare con cifre da 8 bit
- Il tipo **short** permette di trattare numeri a 16 bit, compresi tra -32768 e 32767: è in assoluto il formato meno usato in Java
- Il tipo **long** a 64 bit, che permette di trattare numeri compresi tra -9.223.327.036.854.775.808 ed 9.223.327.036.854.775.807. è necessario posporre al numero la lettera 'L'

Lezione 4

Numeri floating point

- Il tipo **float**, a 32 bit, può contenere numeri positivi e negativi compresi tra $1.40129846432481707 \times 10^{-45}$ e $3.4282346638528860 \times 10^{38}$
- Il tipo **double** a 64 bit può lavorare su numeri positivi e negativi tra $4.94065655841246544 \times 10^{-324}$ e $1.79769313486231570 \times 10^{138}$

`float number = 1.56e3F;`

`double bigNumber = 5.23423e102;`

Lezione 4

boolean

- Una variabile booleana può assumere solamente due valori: 'true' o 'false'

```
boolean a = true;
```

```
boolean b = false;
```

- Il ricorso agli operatori relazionali '==', '!=', '>', '<', '>=' e '<=', permette di assegnare ad una variabile booleana il valore di verità di un'espressione.

Ad esempio:

```
boolean b = (a == 10);
```

assegna a b il valore di verità dell'espressione `a == 10`, che sarà 'true' se la variabile 'a' contiene il valore 10, 'false' in caso contrario

Lezione 4

char

- Il tipo 'char' può contenere un carattere in formato Unicode che comprende decine di migliaia di caratteri, vale a dire gli alfabeti più diffusi nel mondo.
- I valori da 0 a 127 corrispondono, per motivi di retro compatibilità, al set di caratteri ASCII
- Il tipo char è un intero a 16 bit privo di segno: pertanto esso può assumere qualunque valore tra 0 e 65535

char carattere = 'a';

Lezione 4

char

- Il carattere speciale '\' ha il ruolo di specificare alcuni caratteri che altrimenti non è possibile specificare con la tastiera

'\n'	nuova linea
'\r'	a capo
'\f'	nuova pagina
'\"'	carattere apice
'\"'	carattere doppio apice
'\\'	carattere backslash
'\b'	backspace
'\t'	carattere di tabulazione

Lezione 4

Casting: up and down

- **up-casting:** se la variabile destinazione è più capace di quella di partenza

```
byte b = 100;
```

```
short s = b; // promozione da byte a short
```

```
int i = s; // promozione da short a int
```

```
long l = i; // promozione da int a long
```

- **down-casting:** se la variabile destinazione è meno capace di quella di partenza

```
long l = 100;
```

```
int i = (int)l; // cast da long a int
```

```
short s = (short)i; // cast da int a short
```

```
byte b = (byte)s; // cast da short a byte
```

Lezione 4

Razzo Arienne prima del decollo



Lezione 5

Costrutti decisionali ed iterativi

- **Un costrutto decisionale** permette al programmatore di vincolare l'esecuzione di un'istruzione (o di un blocco di istruzioni) ad una condizione booleana
 1. **condizione booleana** è un'espressione della quale si può dire se sia vera o falsa
 2. **blocco di istruzioni** è un insieme di istruzioni racchiuso tra parentesi graffe, che vengono trattate dal compilatore Java come se fossero un'istruzione unica

Lezione 5

Costrutti decisionali

- **if-else** : verificano diverse espressioni e quando viene incontrata l'espressione che restituisce true viene eseguito un determinato pezzo di codice.
- **switch-case**: viene esaminata UNA SOLA espressione, però a seconda del suo risultato cambia il pezzo di codice che verrà eseguito.

Lezione 5

if - else

- Il costrutto condizionale più usato in Java è l'if, che può essere usato nelle due varianti con o senza else

```
if ( condizioneBooleana )  
    istruzione;
```

- La variante con l'else ha una forma del tipo

```
if ( condizioneBooleana )  
    istruzione1;  
else  
    istruzione2;
```

Lezione 5

if - else

- Se vogliamo che venga eseguita più di una istruzione dobbiamo ricorrere ad un blocco:

```
▫ if ( condizioneBooleana ) {  
    istruzione1a;  
    istruzione2a;  
    istruzione3a;  
}  
else {  
    istruzione1b;  
    istruzione2b;  
    istruzione3b;  
}
```

Lezione 5

if - else

- Il costrutto if può comparire anche all'interno di un altro costrutto if, creando strutture nidificate anche molto complesse:

```
▫ if( x >= 0 )  
    if( x <= 10 )  
        System.out.println("x compreso 0 e 10");
```

- Se a questo punto inserisco un else dopo queste istruzioni.

```
if( x >= 0 )  
    if( x <= 10 )  
        System.out.println("x è compreso tra 0 e  
10");  
    else  
        System.out.println("x è maggiore di 10");
```

- A quale dei due if l'istruzione else farà riferimento?

Lezione 5

if - else

- Se ora aggiungiamo un else a quale if si riferisce?

```
if( x >= 0 )  
    if( x <= 10 )  
        System.out.println("x compreso 0 e  
10");  
    else  
        System.out.println("x maggiore di 10");  
else  
    System.out.println("x minore di 0");
```

- Se volessimo forzare un solo else a far riferimento ad un if esterno?

Lezione 5

if - else

- Per creare un blocco di istruzioni

```
if( x >= 0 ) {  
    if( x <= 10 )  
        System.out.println("x compreso 0 e 10");  
}  
else  
    System.out.println("x è minore di 0");
```

Lezione 5

if - else annidati

- E' buona norma di programmazione evitare di ricorrere pesantemente alla nidificazione di istruzioni if, data la confusione che spesso ne segue.

```
if( x >= 0 )
```

```
    if( x <= 10 )
```

```
        System.out.println("x compreso 0 e 10");
```

- può tranquillamente essere sostituito dal seguente, in tutto equivalente:

```
    if( x >= 0 && x <= 10 )
```

```
        System.out.println("x compreso 0 e 10");
```


Lezione 5

if - else nidificati

- E' buona norma di programmazione evitare di ricorrere pesantemente alla nidificazione di istruzioni if, data la confusione che spesso ne segue.

```
if( x >= 0 )  
    if( x <= 10 )  
        System.out.println("x compreso 0 e 10");
```

- può tranquillamente essere sostituito dal seguente, in tutto equivalente:

```
if( x >= 0 && x <= 10 )  
    System.out.println("x compreso 0 e 10");
```

Lezione 5

if - else concatenati

- Una combinazione condizionale si ha quando si fa seguire ad un else una if.

```
if( x <= 0 )  
    System.out.println("x <= 0");  
else if( x <= 10)  
    System.out.println("x > 0 e <= 10");  
else if ( x <= 20)  
    System.out.println("x > 10 <= 20");  
else  
    System.out.println("x è maggiore di 20");
```

- Si noti che in questo caso l'ultimo else comprende tutti i casi non considerati dalle precedenti istruzioni.

Lezione 5

Espressioni condizionali

- L'operatore `?` può essere usato all'interno di espressioni matematiche, dove una delle sotto espressioni sia vincolata ad una particolare condizione booleana.

`espressioneBooleana ? espressione1 : espressione2;`

- Esempio:
 - **`y = x < 0 ? 1 : 2;`**
 - **`if (x < 0) y = 1;`**
`else y=2;`

Lezione 5

Istruzioni switch - case

- Il costrutto switch permette di gestire tutte quelle situazioni in cui dobbiamo prendere scelte diverse a seconda del valore di un'espressione

```
switch (espressione) {  
    case val1:  
        istruzione_na; break;  
    case val2:  
        istruzione_nb; break;  
    default:  
        istruzione_ndefault; break;  
}
```

Lezione 5

Istruzioni switch - case

- L'espressione contenuta tra le parentesi dello switch deve essere di tipo intero (int, byte, short o char);
- ogni istruzione case lavora su di un particolare valore, e fornisce una sequenza di istruzioni da eseguire in quella circostanza
- Tale sequenza termina usualmente con l'istruzione break, che forza il computer a uscire dallo switch, senza verificare i valori successivi.
- blocco di default, ovvero una sequenza di istruzioni da eseguire se non si è verificato nessuno dei casi precedenti.

Lezione 5

I Cicli iterativi

- I cicli vengono abitualmente utilizzati in tutti quei casi in cui bisogna eseguire delle attività ripetitive in modo automatica.
- Abbiamo due tipi di cicli:
 - i cicli con contatore o Cicli **For**
 - I cicli contatore servono per svolgere una data attività per un numero determinato di volte
 - i cicli condizionali o Cicli **While, Do-While**
 - i cicli condizionali vengono usati per eseguire un'attività fino a quando una data condizione viene soddisfatta

Lezione 5

Istruzioni while

- La struttura generale del while è
 - **while(condizioneBooleana) {**
 ISTRUZIONE1;
 ISTRUZIONE2;

 }

Lezione 5

Istruzioni while

- Se si desidera che un ciclo venga ripetuto all'infinito, è sufficiente specificare una condizione sempre vera, tipo

```
while(o == o) {  
    ISTRUZIONE1;  
}
```

- oppure

```
while(true) {  
    ISTRUZIONE1;  
}
```


Lezione 5

Istruzioni while

- Se si desidera che il ciclo venga ripetuto un numero prefissato di volte

```
i = 0;  
while(i<=100) {  
    ISTRUZIONI DA RIPETERE  
    i = i + 1;  
}
```

Lezione 5

L'istruzione do-while

- Dall'istruzione do-while a differenza della precedente, controlla il valore della espressione booleana alla fine del blocco di istruzioni. In questo caso quindi il blocco di istruzioni verrà eseguito sicuramente almeno una volta.
- La sintassi di do-while è la seguente

```
do {  
    istruzione;  
}  
while (espressione_booleana) {  
    istruzione;  
};
```

Lezione 5

Istruzioni for

- Ciclo di 10 iterazioni utilizzando il ciclo while:

```
i=0;
while(i<10) {
    faiQualcosa();
    i++;
}
```

- comporta maggior elaborazione da parte del programmatore pertanto è possibile usare l'istruzione for

```
for(init_statement ; conditional_expr ; iteration_stmt) {
    istruzione
}
```

Lezione 5

Istruzioni for

- **init_statement** rappresenta l'inizializzazione della variabile per il controllo del ciclo
- **conditional_expr** l'espressione condizionale
- **iteration_stmt** l'aggiornamento della variabile di controllo
- Il problema presentato precedentemente può essere risolto utilizzando il ciclo for in questo modo:

```
for (int i=0 ; i<10 ; i++)  
    faiQualcosa();
```

Lezione 6

Foreach - enhanced for Loop

- Per semplificare il ciclo for, in java 6 è stato introdotto un ciclo specializzato che consente di ciclare su array e collection.

for(declaration : expression)

- declaration: variabile compatibile con il tipo di elementi della collezione
- expression: collezione di elementi che può essere un array o un metodo che ritorna un array

Lezione 6

Foreach - enhanced for Loop

- Differenze tra for e for enhanced:

```
int[] a = {1, 2, 3, 4};
```

```
for (int x = 0; x < a.length; x++) {  
    System.out.print(a[x]);  
}
```

```
for (int n : a) {  
    System.out.print(n);  
}
```

Lezione 6

Foreach - enhanced for Loop

- Utilizzo del for enhanced per array annidati:

```
int[][] twoDee = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};  
for (int[] x : twoDee) {  
    for (int y : x) {  
        System.out.println(y);  
    }  
}
```

Lezione 6

VarArgs

- A partire da java 5, è possibile creare metodi con un numero variabile di parametri dello stesso tipo:

void doStuff(int... x) { } //zero o più interi

void doStuff2(char c, int... x) { } //1 char + zero o più interi

void doStuff3(Animal... animal) { } // zero o più tipi Animal

void doStuff4(int x...) { } // sintassi errata

void doStuff5(int... x, char... y) { } // troppi var-args

void doStuff6(String... s, byte b) { } // var-arg x ultimo

Lezione 7

Array

- Molto spesso nei programmi si ha l'esigenza di manipolare un gruppo di variabili dello stesso tipo che contengono valori tra loro correlati.
- Un array è uno strumento concettualmente simile ad una tabella, che accomuna sotto un unico nome un insieme di variabili dello stesso tipo:
- Esempio:

```
int[] temp = new int[144];
```

Lezione 7

Dichiarazione Array

- Come per le variabili semplici dobbiamo indicare un tipo ed un nome, con la differenza che, dopo aver specificato il tipo, è necessario postporre una coppia di parentesi quadre.

```
int[] vettoreDiInteri;
```

- La variabile 'vettoreDiInteri' appena dichiarata non è un vettore, ma solamente un reference ad un vettore

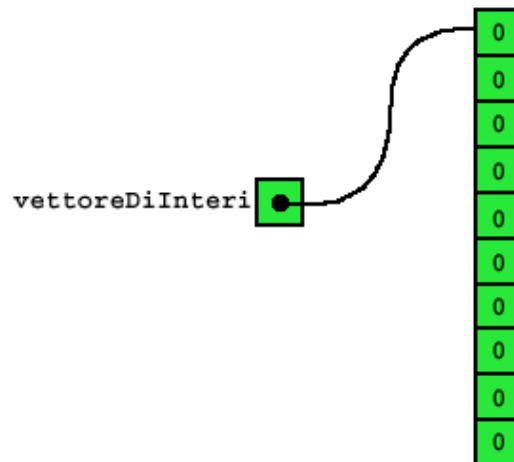
Lezione 7

Assegnamento Array

- Per creare un vettore dobbiamo ricorrere alla parola riservata 'new',

```
vettoreDiInteri = new int[10];
```

- Il valore specificato tra parentesi quadre è la dimensione del vettore
- Un vettore è un oggetto di memoria composto da un certo numero di elementi, ognuno dei quali può contenere un valore



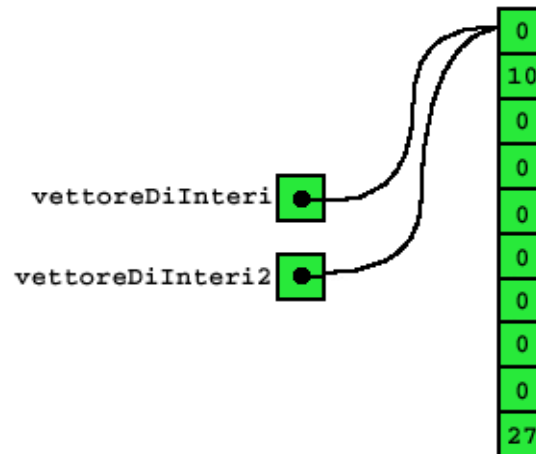
Lezione 7

Dereferenziazione di un array

- La dereferenziazione è un'operazione in cui creiamo una variabile reference che punta ad un array già esistente

```
int[] vettoreDiInteri;  
vettoreDiInteri2 = vettoreDiInteri;
```

- Le modifiche apportate all'oggetto si riflettono anche sull'altro
- Per creare veramente un nuovo array occorre copiare il contenuto in un



Lezione 7

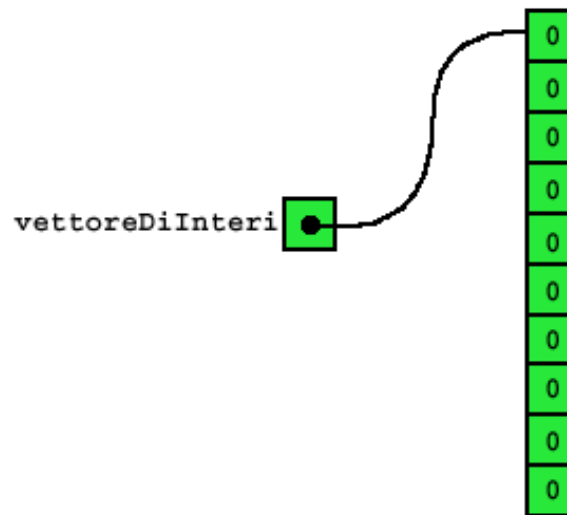
Popolamento di un array

- l'operazione che permette di assegnare un valore ad un elemento del vettore.
- Occorre specificare il nome del vettore seguito dal numero dell'elemento tra parentesi quadre:

`vettoreDiInteri[1] = 10;`

- Gli elementi di un array si con

`vettoreDiInteri[9] = 27;`



Lezione 7

Inizializzazione di un array

- Un vettore può essere inizializzato con una serie di valori, in modo simile a come si può fare con le variabili.

- L'istruzione:

```
int[] vettore = {10,12,14,16,18};
```

- equivale alla sequenza:

```
int[] vettore = new int[5];  
vettore[0] = 10;  
vettore[1] = 12;  
vettore[2] = 14;  
vettore[3] = 16;  
vettore[4] = 18;
```

Lezione 7

Array bi-dimensionali

- Il linguaggio Java consente di creare array bi-dimensionali
- Gli array bidimensionali sono concettualmente simili ad una tabella rettangolare, dotata di righe e colonne.

- Esempio:

```
int i[][] = new int[10][15];
```

- è possibile definire vettori con un numero qualunque di dimensioni:

```
int v1[][][] = new int[10][15][5];
```

```
int v2[][][][] = new int[10][15][12][5];
```

- Tali strutture, in ogni caso, risultano decisamente poco utilizzate.
- I vettori n-dimensionali vengono implementati in Java come array di array.

Array non rettangolari

- possibile la realizzazione di tabelle non rettangolari

```
int tabella[][] = new int[5][];
```

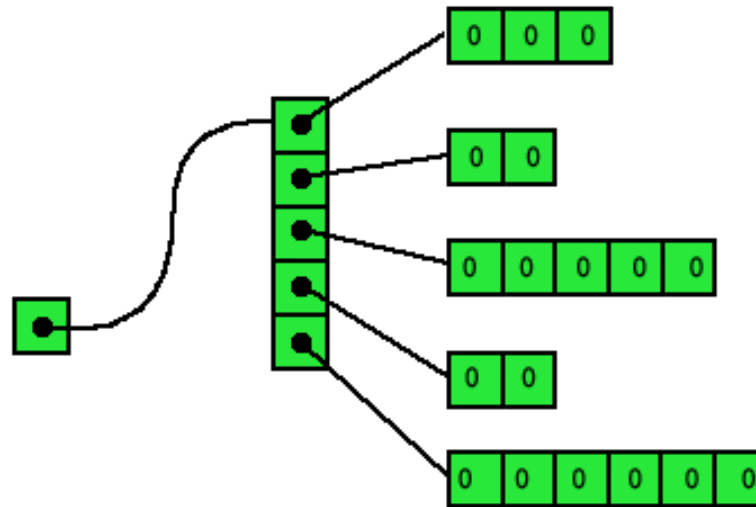
```
tabella[0] = new int[3];
```

```
tabella[1] = new int[2];
```

```
tabella[2] = new int[5];
```

```
tabella[3] = new int[2];
```

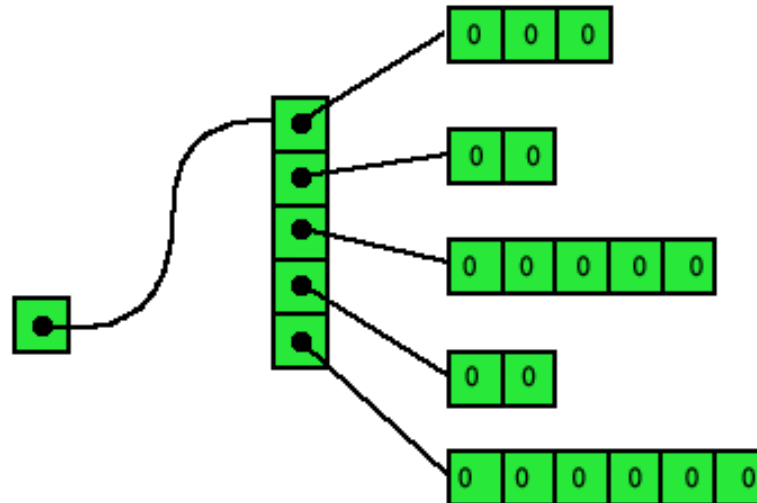
```
tabella[4] = new int[6];
```



Inizializzazione di array n-dimensionali

- Un Array multidimensionale può essere inizializzato con una serie di valori, in modo simile a come si può fare con gli Array semplici.

```
int[] vettore = { { 10,12,14},{16,18},{20,22,24,26}};
```



Lezione 8

Enumeration

- In Java 5 è possibile definire una variabile con un numero predefinito di valori.
- Tale variabile può essere definita sia come variabile nella classe e sia come classe esterna.

- **enum Size { SMALL, BIG;}**

Caratteristiche:

- Hanno modificatori: public static final
- Ereditano metodi:
 - **public static MyEnum valueOf(String name)**
 - **public static MyEnum[] values()**

Lezione 8

Enumeration

Una enum può essere scomposta in qualcosa del genere:

```
public class Size {  
  
    public static final Size BIG = new Size("BIG", 0);  
    public static final Size SMALL = new Size("SMALL", 0);  
  
    private String name;  
    private int index;  
  
    public Size(String name, int index) {  
        this.name = name;  
        this.index = index;  
    }  
  
}
```

Lezione 8

Enumeration

I metodi `valueOf` e `values` possono essere così rappresentati:

```
public class Size {  
    ....  
    public static Size valueOf(String name) {  
        Size ret = null;  
        if (BIG.name.equals(name)) {  
            ret = BIG;  
        } else if (SMALL.name.equals(name)) {  
            ret = BIG;  
        }  
        return ret;  
    }  
    public static Size[] values(){  
        Size[] size = {BIG, SMALL};  
        return size;  
    }  
}
```

Lezione 8

Enumeration

Le enum possono possedere costruttori, metodi e sovrascriverli:

```
public enum Size{  
    SMALL(7), BIG(15) {  
        @Override  
        public String getCode() {  
            return "A";  
        }  
    };  
    private int number;  
  
    Size(number) {  
        this.number = number;  
    }  
  
    public String getCode() {  
        return "B";  
    }  
}
```

Lezione 9

Le stringhe alfanumeriche

- In Java si possono utilizzare e gestire le stringhe tramite le due classi:
 - **String**: un oggetto di tipo String è una stringa costante
 - **StringBuffer**: un oggetto di tipo StringBuffer è una stringa variabile nel contenuto e nella dimensione. Meno “economico” di String.

Lezione 9

Le stringhe alfanumeriche

- La creazione di un oggetto String può essere:
 1. **Esplícita:** attraverso il costruttore polimorfico `String()`
Es. `String S= new String("parola");`
 2. **Implicita:** quando il compilatore incontra una serie di caratteri delimitati da virgolette crea un oggetto di tipo String a cui assegna la stringa individuata.
Es. `String S = "parola";`

Lezione 9

Le stringhe alfanumeriche

In riferimento a: `String s = "parola";`

- `length()`: ritorna la lunghezza della stringa
Es: `int len= s.length(); // len== 6`
- `charAt(int i)`: ritorna il carattere in posizione i-esima
Es: `char c=s.charAt(0) // c == 'p'`
- `indexOf(char c)`: ritorna l'indice della prima occorrenza del carattere indicato
Es: `int i=s.indexOf('o') // i == 3`
- `substring(int da,int a)`: ritorna una sottostringa
Es: `String sub = s.substring(2,4) // sub=="ro"`

Lezione 9

Le stringhe alfanumeriche

Concatenazione

- E' possibile concatenare stringhe tramite l'operatore +

```
String s = "Prova " + "di" + " concatenazione";  
System.out.println(s);
```

- Output: Prova di concatenazione

Modulo 3

Object Oriented programming

A series of horizontal lines in teal and white, stacked and offset to the right, creating a modern, layered effect.

Lezione 1

Classi e oggetti

- La classe modella un insieme di oggetti omogenei (le istanze della classe) ai quali sono associate proprietà statiche e dinamiche
- Ogni classe è descritta da:
 - Nome della classe (l'identificatore)
 - Attributi (lo stato)
 - Operazioni (il comportamento)

Lezione 1

Classi e oggetti

Importanza dell'identificativo

- Due oggetti con identificativi distinti sono comunque distinti, anche se hanno i valori di tutti gli attributi uguali
- Due oggetti diversi devono avere identificativi diversi, anche se possono avere gli stessi valori per tutti gli attributi

Lezione 2

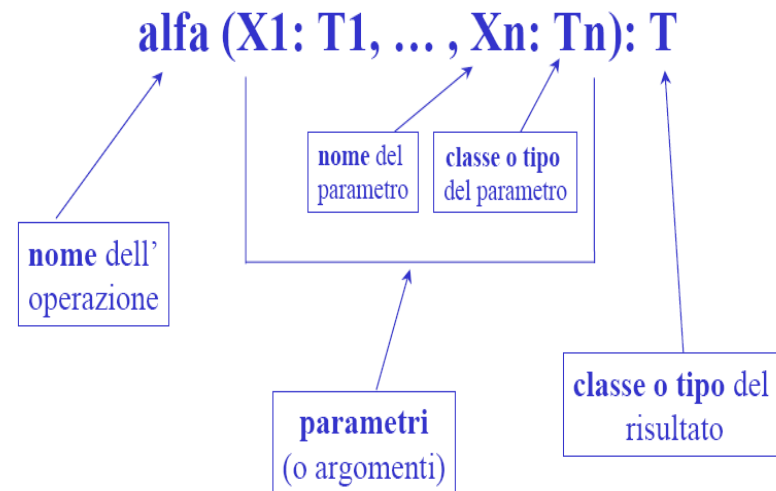
Attributi e metodi

- Un attributo
 - modella una proprietà della classe
 - valida per tutte le istanze
- Un attributo può considerarsi una funzione che associa un valore di un certo tipo ad ogni oggetto che è istanza della classe

Lezione 2

Attributi e metodi

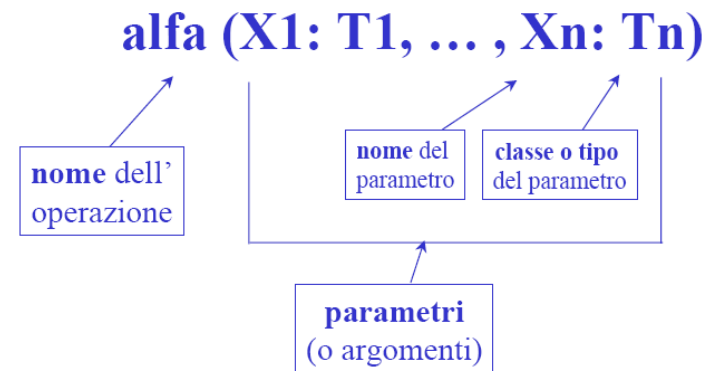
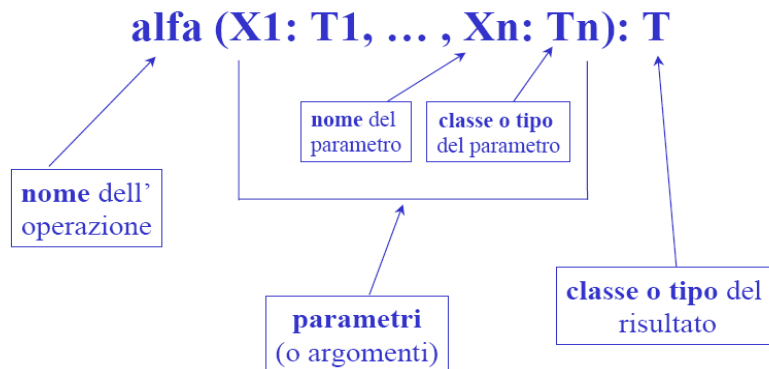
- Le classi sono caratterizzate anche da proprietà dinamiche, che in java si chiamano metodi.
- Un metodo associato ad una classe C indica che sugli oggetti della classe C si può eseguire una computazione
 - per calcolare una proprietà,
 - per effettuare cambiamenti di stato



Lezione 2

Attributi e metodi

- In una classe, un metodo si definisce specificando i parametri e il tipo del risultato (se c'è).



- Non è necessario che un metodo restituisca un valore o un oggetto. Un metodo può anche effettuare solamente azioni senza calcolare un risultato.

Lezione 2

Attributi e metodi

- I metodi sono dei blocchi di istruzioni definiti in una classe
- Sono di **due** tipologie:
 - **Funzione:** elaborano i dati e restituiscono un valore
 - **Procedure:** eseguono la procedura e non restituiscono un valore

Lezione 2

Attributi e metodi

- **Metodi funzione:** elaborano eventuali parametri e restituiscono un risultato:

```
int moltiplicavalori(int a, int b) {  
    risultato= a * b;  
    return(risultato);  
}
```

Lezione 2

Attributi e metodi

1. **int** è il tipo di valore che la funzione restituirà (quindi il tipo della variabile risultato);
2. **moltiplicavalori** è il nome della funzione;
3. **argomenti**: all'interno delle parentesi tonde troviamo i due argomenti che forniamo alla funzione con i loro relativi tipi
4. **blocco di istruzioni** : all'interno delle parentesi graffe troviamo il blocco di istruzioni della funzione
5. **return**: l'istruzione `return(...)` è quella che permette alla funzione di restituire il valore risultato. Le istruzioni che seguono l'istruzione `return` vengono ignorate

Lezione 2

Attributi e metodi

- **Metodi procedura:** elaborano eventuali parametri e non restituiscono un valore. Servono per evitare di ripetere un blocco di codice

```
void nomeMetodo(int a, int b) {  
    System.out.println(a+" - "+b );  
}
```

Lezione 2

Attributi e metodi

- **nomeMetodo:** è il nome della funzione;
- **argomenti:** all'interno delle parentesi tonde troviamo i due argomenti che forniamo alla funzione con i loro relativi tipi
- **blocco di istruzioni:** all'interno delle parentesi graffe troviamo il blocco di istruzioni della funzione
- **void** (vuoto): indica che la funzione non restituirà nessun valore ma eseguirà solamente le istruzioni indicate
- **l'istruzione:** `System.out.println(...)` è quella che permette alla funzione di visualizzare a console i parametri passati

Lezione 2

Attributi e metodi

- Le proprietà sono collegate ai tipi, che posso essere :
 - **primitivi**: si prestano ad un uso intuitivo
 - **reference**: fanno riferimento ad oggetti, array, ecc
- A seconda della modificabilità, possono essere.
 - **Costanti** (variabili final): dati che non possono subire alcuna modifica
 - **Variabili** : dati suscettibili di modificazione
- A seconda della globalità possono essere:
 - **static** : le modifiche apportate da un oggetto a questi dati si propagano a tutte le istanze di quell'oggetto
 - **di istanza**: le modifiche apportate non si ripercuotono alle altra istanze

Lezione 3

Costruttori

- Il costruttore è un metodo che viene eseguito quando si istanzia una classe.

```
public class MiaClasse
    public MiaClasse() {
        //blocco di istruzioni
    }
}
```

- **public:** è necessario che sia accessibile per istanziare la classe
- **MiaClasse:** ha lo stesso nome della classe
- **blocco di istruzioni:** sono le istruzioni che implementiamo

Possono esistere più metodi costruttori con diversi tipi e quantità di parametri (overloading)

Lezione 4

Modificatori e costanti

```
<modificatore>[static][final]<tipoRitornato><nomeMetodo>  
([<parametri>])  
{ <body> }
```

Metodi: sintassi completa

```
<modificatore>[static][final]<tipo><identificatore>[=<valore>]  
[,<identificatore>[=<valore>]...]
```

Variabili: sintassi completa

<modificatore>	Descrizione
public	<ul style="list-style-type: none">● accessibile ai metodi della classe in cui è usato● accessibile ai metodi di altre classi
protected	<ul style="list-style-type: none">● accessibile ai metodi della classe in cui è usato● accessibile ai metodi delle sottoclassi della classe in cui è usato● accessibile ai metodi di classi dello stesso package della classe in cui è usato
	<ul style="list-style-type: none">● accessibile ai metodi della classe in cui è usato● accessibile ai metodi di classi dello stesso package della classe in cui è usato
private	<ul style="list-style-type: none">● accessibile ai metodi della classe in cui è usato

Lezione 5

Incapsulamento dei dati

- L'incapsulamento è una tecnica mediante la quale si proteggono i dati evitando che si possano manipolare, lasciando però la possibilità di interrogarne il valore.
- In alcuni linguaggi procedurali esiste solo "l'incapsulamento sui tipi primitivi", ma non quello sui nuovi tipi definiti dall'utente.
- Nei linguaggi OOP, invece, è supportato anche l'incapsulamento per i tipi di dati definiti dall'utente
- In altri termini l'incapsulamento consente di legare, in riferimento alla classe, i dati (membri o variabili) e le operazioni (metodi o funzioni) con una limitazione di accesso ad essi.
- Questo modo di operare protetto elimina i problemi di difficoltà di manutenzione

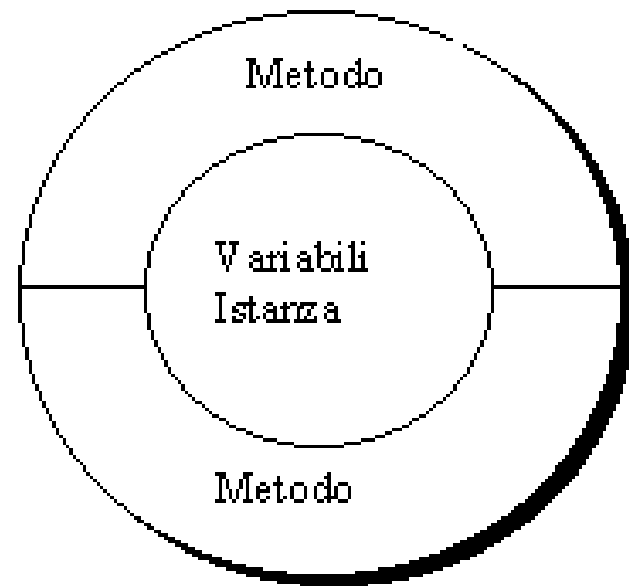
Lezione 5

Incapsulamento dei dati

- In questa classe possiamo notare che la proprietà x privata è accessibile in scrittura dal metodo scrivo() ed in lettura dal metodo leggo()

- Esempio:

```
class MiaClasse {  
    private int x;  
    public int leggo() {  
        return x;  
    }  
    public void scrivo(int x) {  
        this.x = x;  
    }  
}
```



Lezione 6

L'Overloading dei metodi

- Per riferirsi agli oggetti ed ai metodi si utilizzano dei nomi.
- Nomi scelti con criterio rendono più semplice la comprensione del codice.
- Spesso la stessa parola esprime differenti significati: è **overloaded**.

Lezione 6

L'Overloading dei metodi

- In Java c'è la possibilità di utilizzare lo stesso nome per fare riferimento a più metodi
- Il metodo da eseguire viene stabilito in funzione del contesto
- Il contesto è indicato dagli argomenti e dal valore di ritorno del metodo
- Tipicamente il metodo costruttore di una classe è overloaded, visto che spesso è utile stabilire diversi modi di inizializzazione dell'oggetto

Lezione 6

L'Overloading dei metodi

- Esempio di overloading del costruttore:

```
class Esempio {  
    int prop;  
    public Esempio() { codice };  
    public Esempio(int prop) {  
        this.prop=prop;  
    }  
}
```

- L'argomento identifica **univocamente** il metodo
- In questo esempio anche prop è overloaded. Con il riferimento this si risolve l'ambiguità.

Lezione 6

L'Overloading dei metodi

- Esempio di overloading di un metodo:

```
class Esempio {  
    int prop;  
    public void metodo() { codice };  
    public int metodo() {  
        return prop;  
    }  
}
```

- I valori di ritorno identificano **univocamente** i metodi

Modulo 4

Introduzione alla progettazione con UML



Cos'è UML ?

- UML (Unified Modeling Language) è un linguaggio standard di modellazione per
 - Specificare
 - Visualizzare
 - Costruire
 - Documentare
- domini applicativi eterogenei, adatto maggiormente a progettare sistemi object-oriented e sistemi component-based.

UML è un linguaggio...

- UML è un linguaggio pertanto costituito da sintassi e semantica
 - **sintassi**: regole attraverso le quali gli elementi del linguaggio (parole) sono assemblate in espressioni (frasi).
 - **semantica**: regole attraverso le quali alle espressioni sintattiche viene assegnato un significato.

UML non è una metodologia!

- La metodologia serve per specificare:
 - quale linguaggio di modellazione utilizzare per descrivere il lavoro progettuale
 - quali sono i passi necessari per raggiungere la “industrializzazione” nella produzione del software.

UML non è un processo!

- Il processo è un insieme di regole che definisce come un progetto di sviluppo dovrebbe essere condotto.
 - Include una descrizione e sequenzializzazione di attività, documenti e modelli.
 - Definisce i criteri per il monitoraggio delle attività e la valutazione degli artefatti.

...standard...

- Per standard si intende:
 - un paradigma codificato per la produzione di tecnologie fra loro compatibili e interoperabili riferiti ad hardware, software o infrastrutture di rete.

...di modellazione...

- Modellare significa descrivere un sistema in termini di:
 - entità coinvolte
 - relazioni esistenti tra di loro
- Esempio:
 - Diagrammi di flusso
 - UML
 - Diagrammi Entita-Relazione

La qualità di un modello

- **Accuratezza:**
 - deve descrivere il sistema correttamente, completamente e senza ambiguità;
- **Consistenza:**
 - le diverse viste devono completarsi vicendevolmente per formare un insieme coerente
- **Semplicità:**
 - deve poter essere compreso, senza troppi problemi, da persone estranee al processo di modellazione;
- **Manutenibilità:**
 - la variazione dello stesso deve essere la più semplice possibile.

... per specificare, visualizzare, costruire e documentare ...

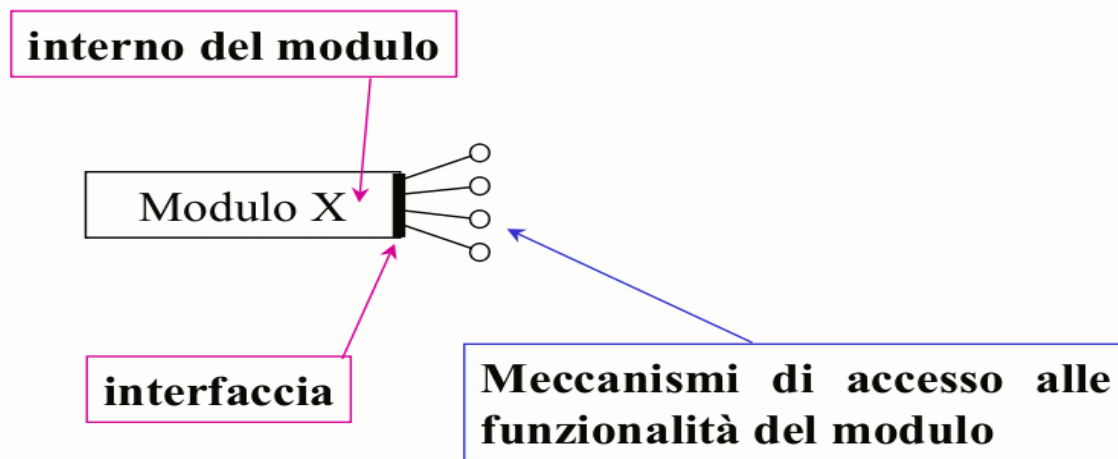
- Specificare
 - Dettagli di implementazione
- Visualizzare
 - Un immagine è meglio di 100 parole
- Costruire
 - Idee, pensieri
- Documentare
 - Interazione con gruppi esterni

...domini applicativi eterogenei...

- Dominio eterogenei
 - sanità, finanza, TLC, aerospazio
 - indipendentemente dalla piattaforma
- Sistema:
 - una singola organizzazione vista nella sua globalità (es. azienda)
 - una parte di un'organizzazione (es. divisione, oppure processo)
 - un insieme di organizzazioni, o di parti di organizzazioni, in relazione tra loro (es. processi di interazione Business-to-Business)

...e sistemi component-based.

- I quattro “dogmi” della modularizzazione sono:
 - Alta coesione (omogeneità interna)
 - Information hiding (poco rumore nella comunicazione)
 - Basso accoppiamento (indipendenza da altri moduli)
 - Interfacciamento esplicito (chiare modalità d’uso)



...e sistemi component-based...

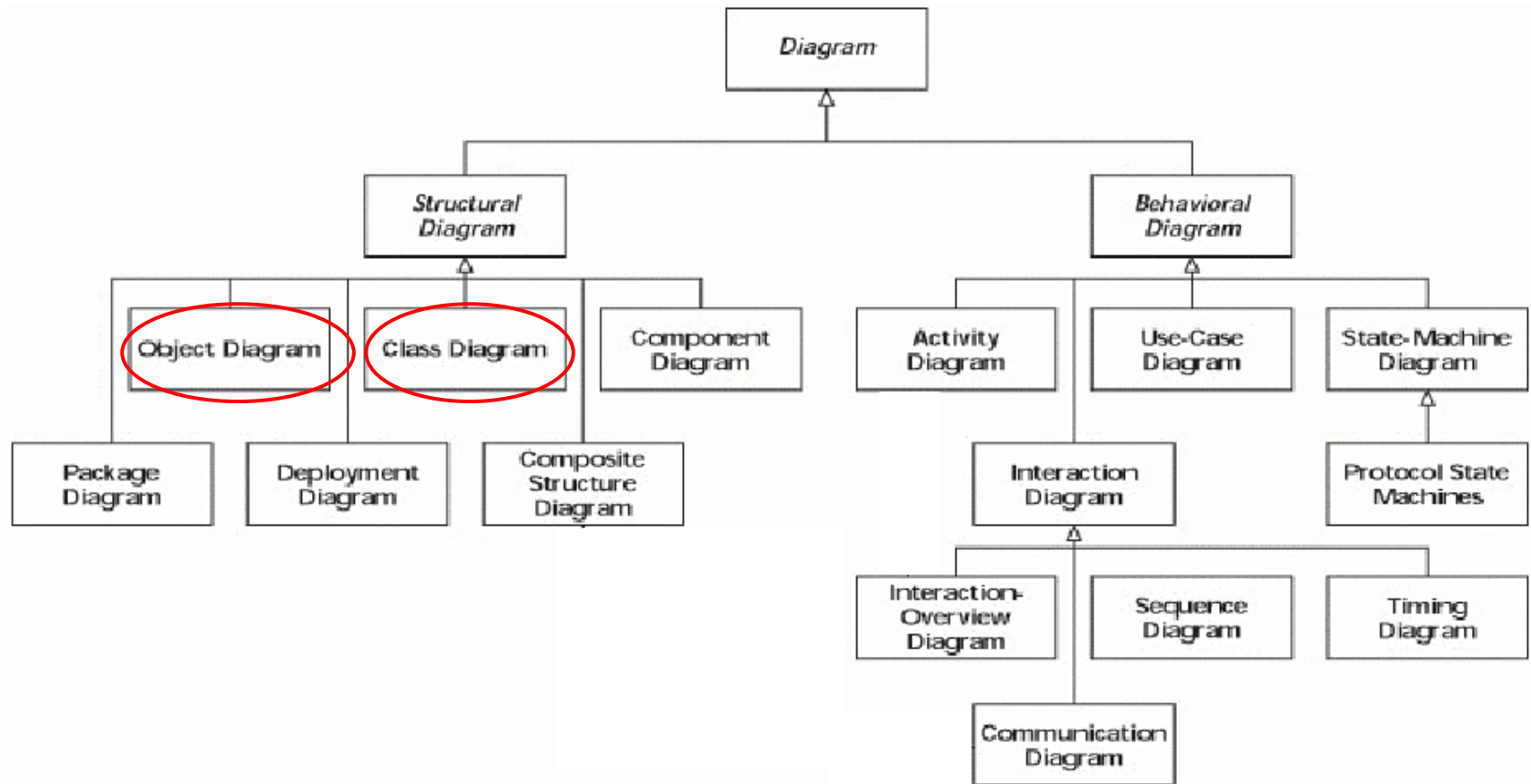
- Il PC che stiamo utilizzando ne è un esempio...
 - Componenti riutilizzabili
 - Dividere la logica dall'interfaccia
 - Utilizzare uno standard di comunicazione

...adatto maggiormente a progettare sistemi object-oriented.

- I concetti OO si sono sviluppati dal 1970 attraverso diversi linguaggi di programmazione c++, smalltalk, java, eiffel.

Lezione 3

Diagramma delle classi

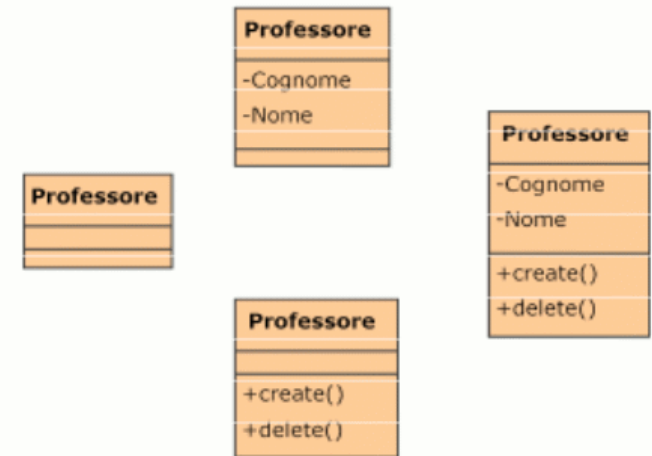
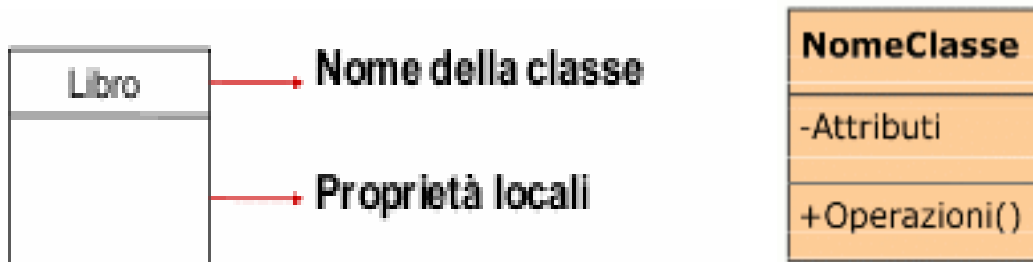


Class Diagram

- E' il diagramma più importante in UML perché definisce gli elementi base del sistema
- Rappresenta il sistema attraverso una visione statica
- Rappresenta le classi che compongono il sistema ed i relativi attributi e comportamenti
- Specifica, mediante le associazioni, i vincoli che legano tra loro le classi

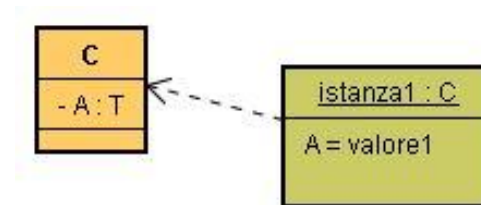
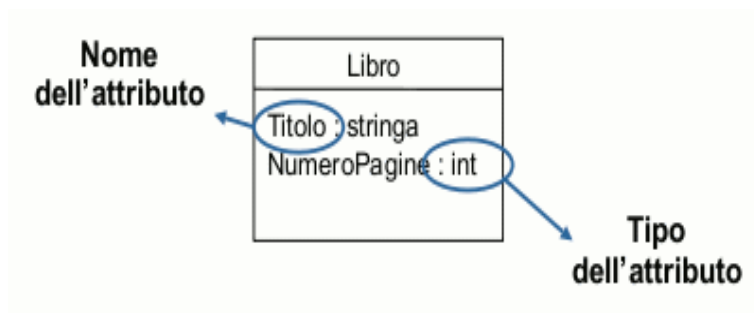
Classi

- La classe modella un insieme di oggetti omogenei (le istanze della classe) ai quali sono associate proprietà statiche e dinamiche
- Ogni classe è descritta da:
 - Nome della classe (l'identificatore)
 - Attributi (lo stato)
 - Operazioni (il comportamento)



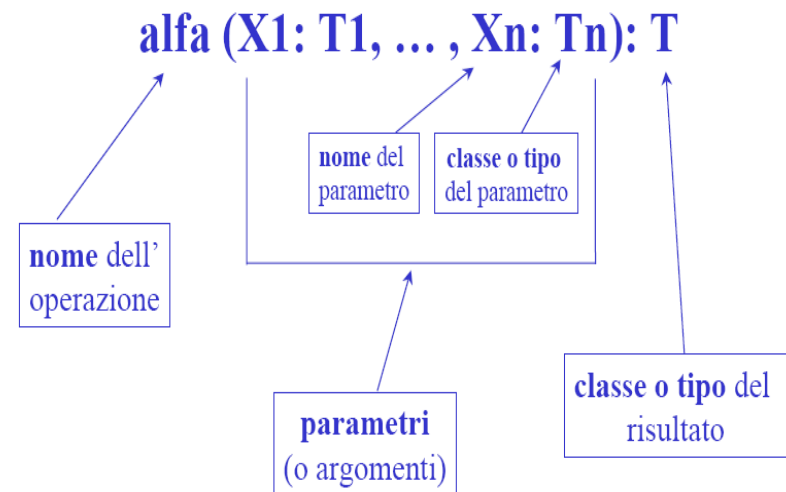
Attributi di Classe

- Un attributo
 - modella una proprietà della classe
 - valida per tutte le istanze
- Formalmente, un attributo A della classe C può considerarsi una funzione che associa un valore di tipo T ad ogni oggetto che è istanza della classe C



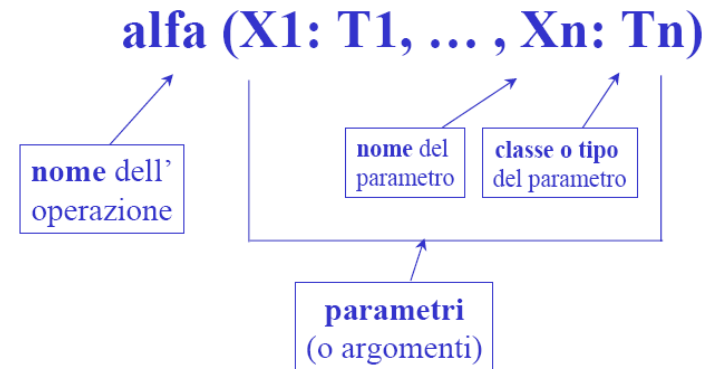
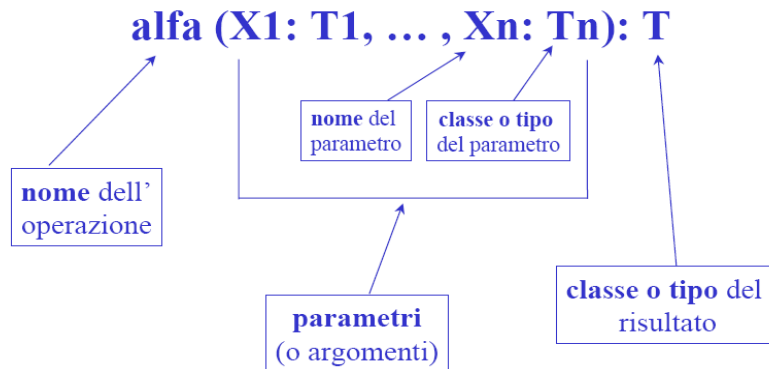
Operazione di Classe

- In realtà, le classi (e quindi le loro istanze) sono caratterizzate anche da proprietà dinamiche, che in UML si chiamano operazioni. Un'operazione associata ad una classe C indica che sugli oggetti della classe C si può eseguire una computazione (detta anche metodo),
 - per calcolare una proprietà,
 - per effettuare cambiamenti di stato
- In una classe, un'operazione si definisce specificando i parametri e il tipo del risultato (se c'è).



Operazione di Classe

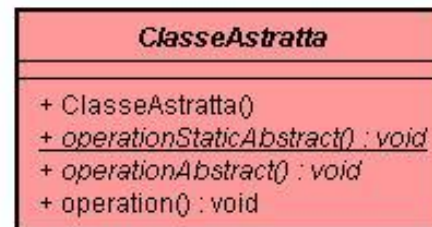
- In una classe, un'operazione si definisce specificando i parametri e il tipo del risultato (se c'è).



- Non è necessario che un'operazione restituisca un valore o un oggetto. Un'operazione può anche effettuare solamente azioni senza calcolare un risultato.

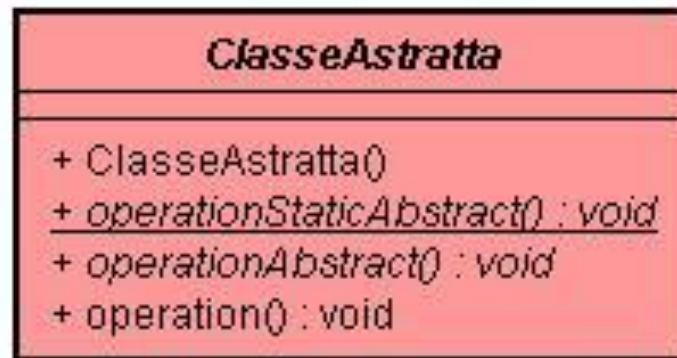
Classe Astratta

- E' una classe che non ha una concreta implementazione ma ha solo la dichiarazione delle operazioni
- Non può essere istanziata, ma può avere costruttori.
- Può contenere sia dei metodi astratti (statici e di istanza) sia dei metodi concreti
- Se un metodo è definito astratto, anche la classe deve essere astratta,
- Se nessun metodo è astratto la classe può comunque essere astratta.



Classe Astratta

- Il nome della classe astratta è in *corsivo*
- E' possibile inserire il costruttore (non astratto)
- I metodi statici sono sottolineati
- Il metodi astratti sono in corsivo
- E' possibile indicare metodi “concreti”



Interfaccia

- Una interfaccia “interface” ha una struttura simile a una classe, ma può contenere SOLO:
 - costanti
 - metodi d'istanza astratti
 - metodi e proprietà “public”
- quindi non può contenere:
 - né costruttori
 - né variabili statiche
 - né variabili di istanza
 - né metodi statici
 - metodi e proprietà diversi da “public”

Interfaccia

- Se scriviamo una interaccia in questo modo:

```
interface MiaInterf {  
    int i=0;  
    String metodo();  
}
```

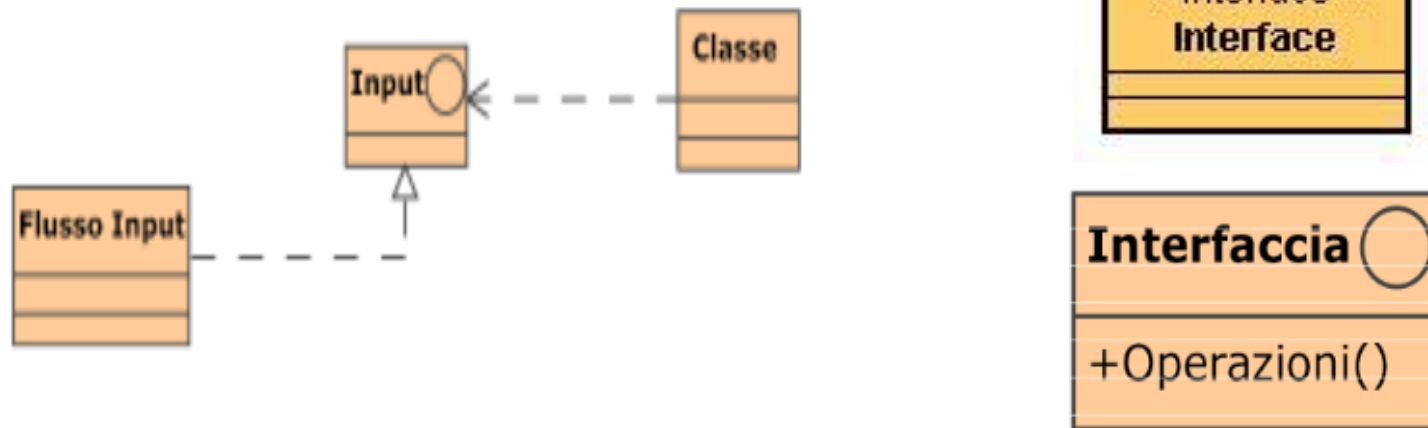
- Il compilatore “aggiungerà” questi modificatori

```
interface MiaInterf {  
    public static final int i=0;  
    public abstract String metodo();  
}
```



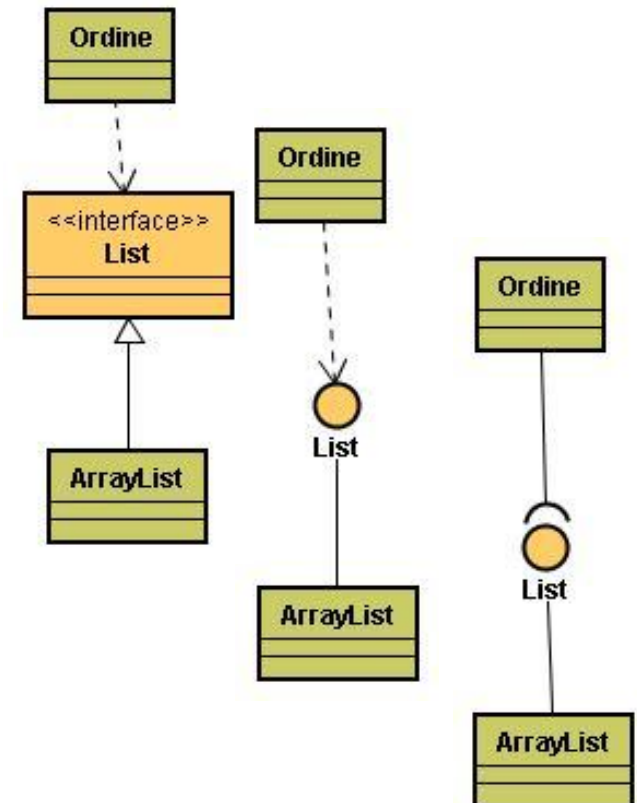
Interfaccia

- La classe provvede una particolare implementazione ma le altre classi potranno interagire con essa usando una interfaccia
- Si aumenta notevolmente l'indipendenza tra le classi



Interfaccia

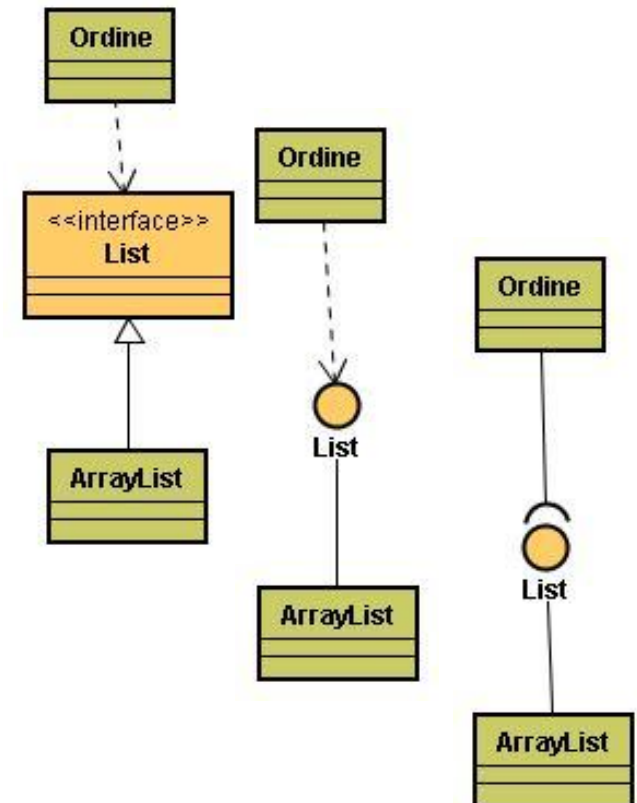
- Possiamo rappresentare lo scenario di un'interfaccia in 3 modi diversi:
 - Uguale alla classe ma con indicato lo stereotipo <<interface>>
 - Con una icona a forma di pallina (lollipop -> “leccalecca”) puntata da una freccia tratteggiata (dipendenza)
 - Con una icona a forma di pallina agganciata da una icona a forma di semicerchio (socket -> “presa”) introdotta con UML 2



Interfaccia

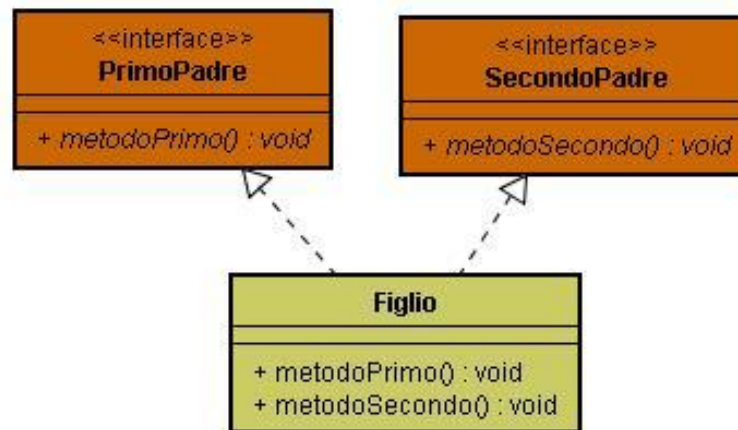
- Ed ecco come possiamo rappresentare il codice nel linguaggio Java:

```
public class Ordine {  
    private List list = new ArrayList();  
}  
  
public interface List {  
}  
  
public class ArrayList implements List {  
}
```



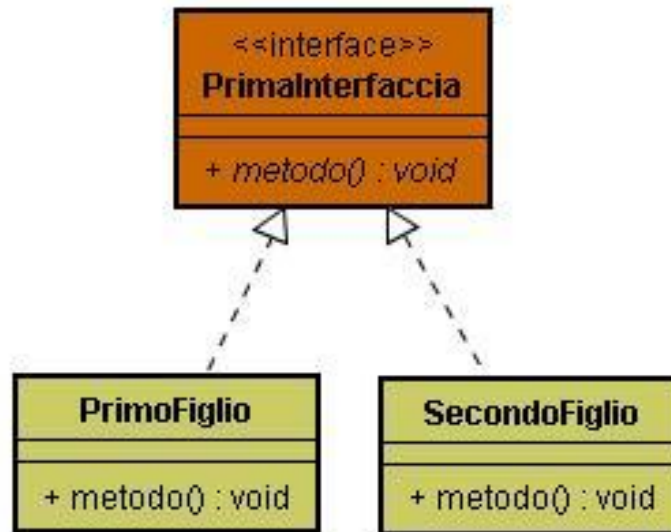
Interfaccia

- Le interfaccia permettono di emulare l'ereditarietà multipla, ma senza i problemi ad essa collegati
- Come le classi, le interfacce possono usare l'ereditarietà con altre interfacce
- Le interfacce specificano un protocollo che deve essere implementato dalle sottoclassi



Interfaccia

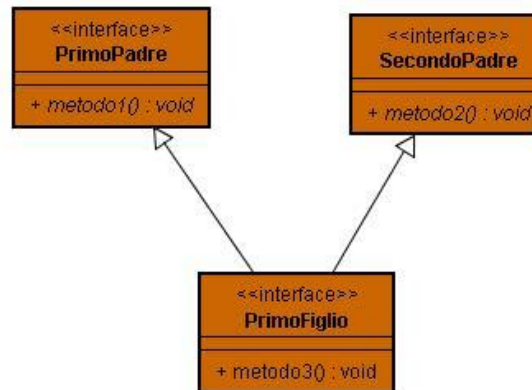
- Inoltre sarà sempre possibile che più classi ereditino da una stessa interfaccia come in figura sottostante.



Interfaccia

- Una interfaccia può estendere una o più interfacce (non classi), indicate dopo la parola chiave `extends`. Per le interfacce non vale la restrizione di "ereditarietà singola" che vale per le classi.

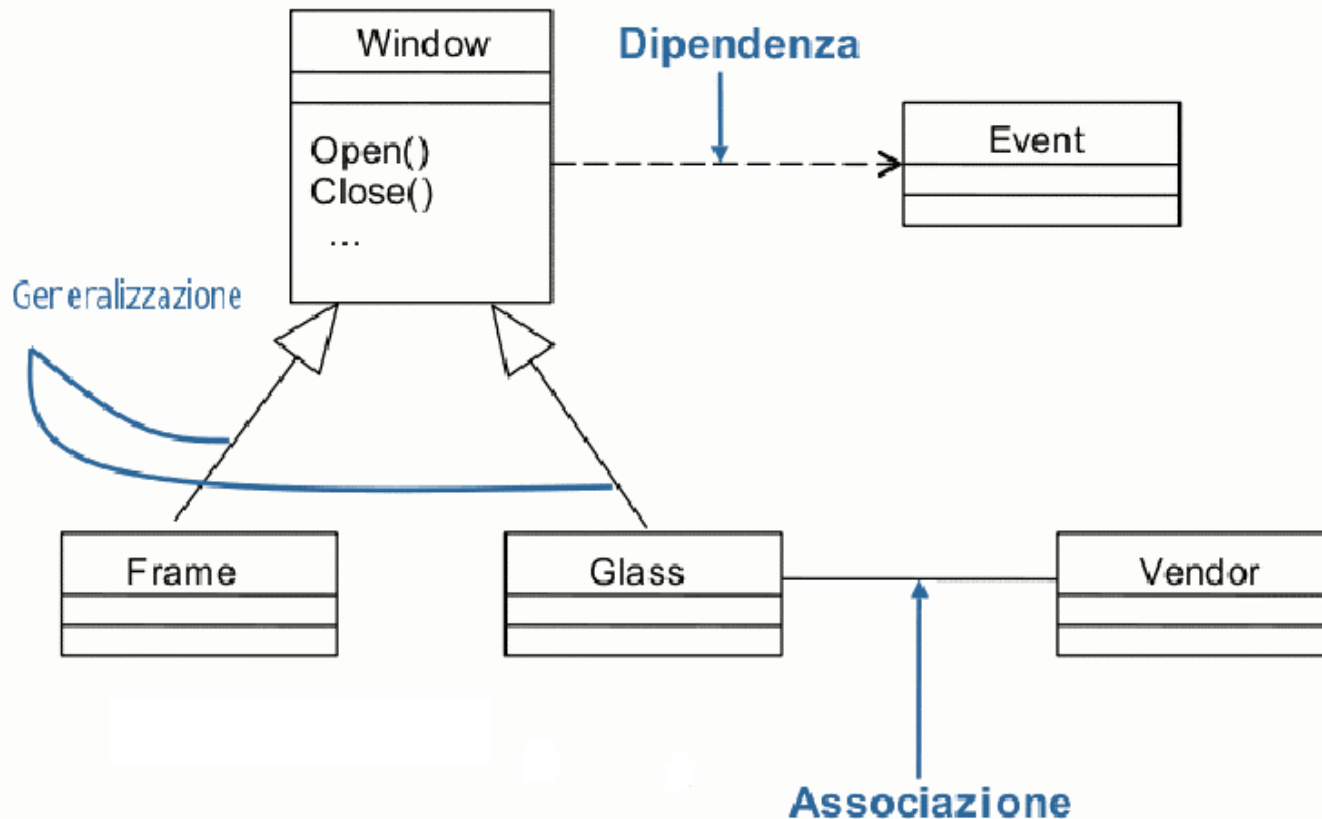
```
public interface PrimoFiglio extends PrimoPadre, SecondoPadre {  
    public void metodo3();  
}
```



Relazioni tra Classi

- Quando si costituiscono i diagrammi delle classi, accade molto raramente che le entità coinvolte siano destinate a rimanere isolate
- Tipicamente, tutte le classi identificate sono connesse con altre secondo precisi criteri
 - Non è sufficiente identificare le classi che compongono il sistema,
 - Ma è necessario individuare anche le eventuali relazioni che le legano
- Nel mondo Object Oriented, tutte le relazioni possono essere facilmente ricondotte ai tre tipi fondamentali
 - Dipendenza
 - Generalizzazione
 - Associazione

Relazioni tra Classi



1-Relazione di dipendenza

- Indica che la variazione dello stato di un'istanza della classe indipendente implica un cambiamento di stato nell'istanza della classe dipendente
- Es. precedente
 - La dipendenza lega le classi “Window” alla classe “Event”
 - Indica che un cambiamento di stato dell'oggetto “Event” genera un aggiornamento dell'oggetto “Window” che lo utilizza
 - Non è vero l'inverso

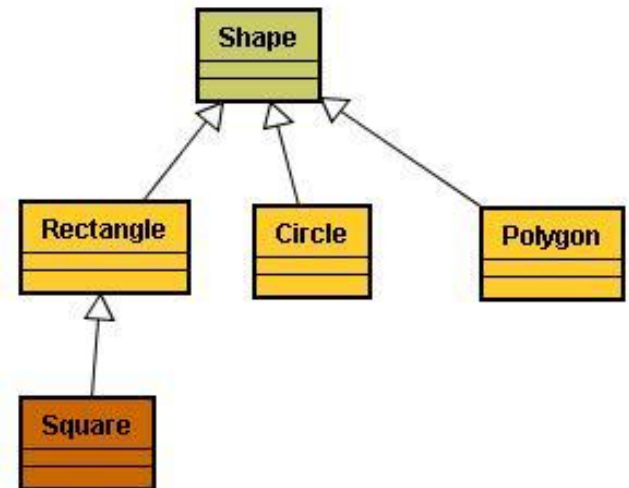
2-Relazione di generalizzazione

- Indica un legame tra una classe generale, denominata super-class o parent, ed una sua versione più specifica, denominata sub-class o childclass
- Una classe figlio specializza il comportamento di quella del padre
- Es. precedente
 - La generalizzazione lega le classi “Window” alla classe ”DialogBox”
 - La classe “DialogBox” va ad aggiungere un comportamento specifico alla classe padre “Window”



2-Relazione di generalizzazione

- Lega un oggetto (detto super-class o parent) ad uno più specifico (detto sub-class o child-class) appartenente alla stessa tipologia
- È spesso indicata con la denominazione Is-kind-of (è di tipo), per specificare che un oggetto specializzato è “del tipo” dell’oggetto più generale
- È possibile asserire che le classi Rectangle, Circle e Polygon sono del tipo della classe Shape
- La classe Square, applicando un ragionamento ricorsivo, è anch’essa di tipo Shape oltre ad essere di tipo Rectangle



3-Relazione di associazione

- Indica il significato per cui un oggetto di una classe è connesso ad un altro oggetto
- È una relazione strutturata tra classi che definisce un canale di comunicazione bi-direzionale fra le classi
 - Tra due classi è associazione binaria
 - Tra n classi è associazione “n-aria”
- Una associazione può avere un nome

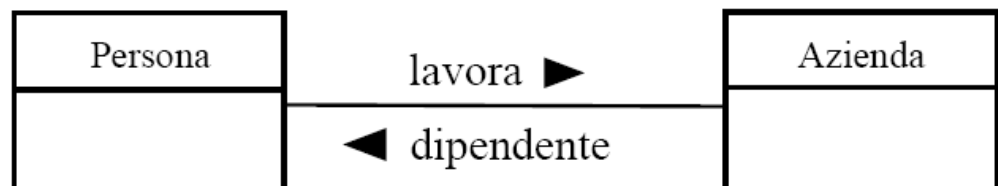


3-Relazione di associazione

- Può avere una direzione per distinguere meglio il verso della comunicazione

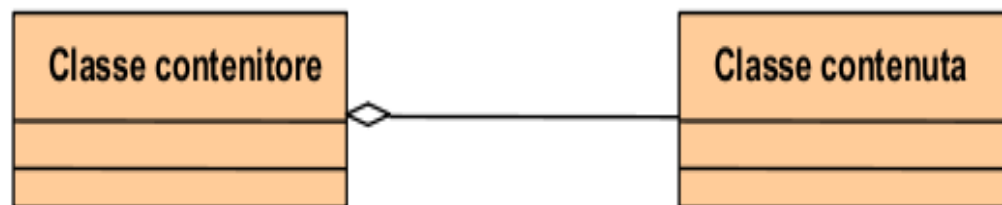


- È anche possibile assegnare due nomi con i relativi versi alla stessa associazione
- Osserviamo ancora che le frecce che simboleggiano il verso non aggiungono nulla al significato della associazione (che è sempre una relazione), ma aiutano ad interpretare il senso dei nomi scelti per l'associazione
- Le frecce che simboleggiano il verso si indicano anche nel link che sono istanze delle associazioni



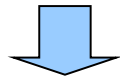
3-Relazione di aggregazione

- Le aggregazioni sono una forma particolare di associazione
- Definisce una associazione del tipo “parte di”
 - Relazioni tra classi in cui: una rappresenta un oggetto (Whole) e l'altra una sua componente (Part)
 - È detta anche Whole-Part
- E' una associazione asimmetrica
- Graficamente
 - Linea piena con rombo rivolto verso la classe più generale



3-Relazione di composizione

- E' una forma di aggregazione di tipo forte
- La loro esistenza ha senso solo in relazione al contenitore
- Creazione e distruzione avvengono nel contenitore
- I componenti non sono parti di altri oggetti
- Se si cancella il contenitore si cancellano anche le parti

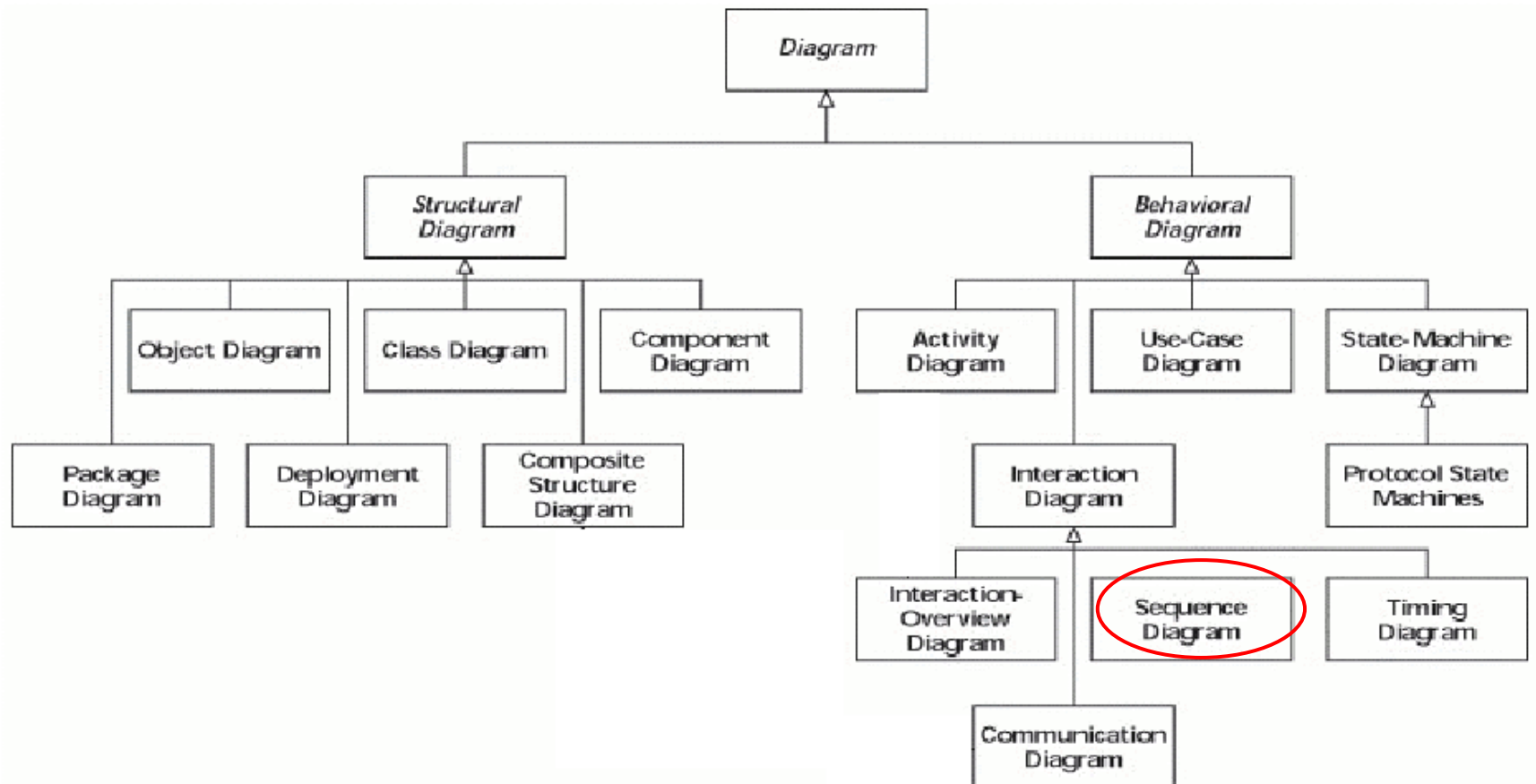


- Le parti componenti non esistono senza il contenitore



Lezione 4

Diagramma di sequenza



Sequence Diagram

- Il Sequence Diagram descrive il comportamento dinamico di un gruppo di oggetti che interagiscono
- Il Sequence Diagram illustra le interazioni degli oggetti ordinate in modo cronologico
- Sono usati per “formalizzare” gli scenari, in termini di:
 - Entità (oggetti)
 - Messaggi scambiati (metodi)
- Evidenzia la sequenza temporale delle azioni in base ad una dimensione: il TEMPO

Sequence Diagram

- Le interazioni tra gli oggetti avvengono in una sequenza specifica
- La sequenza consuma del tempo per andare dall'inizio alla fine
- Un Sequence Diagram contiene:
 - **Oggetti**
 - Graficamente – rettangoli con il nome sottolineato
 - **Messaggi**
 - Graficamente – tramite frecce a linea continua
 - **Tempo**
 - Graficamente – rappresentato progressivamente nell'asse verticale

Sequence Diagram - Elementi

- **Oggetti**

- Rappresenta una singola entità
- In un ambiente object-oriented rappresenta un'istanza
- Può rappresentare una particolare istanza di un oggetto o di un attore



Sequence Diagram - Elementi

- **Messaggio**
 - Rappresenta la comunicazione tra due oggetti
 - Va dalla lifeline di un oggetto alla lifeline di un altro oggetto
 - Tipi di messaggi
 - Semplice – trasferimento di controllo da un oggetto ad un altro
 - Sincrona – chi invia il messaggio deve attendere la risposta
 - Asincrona – chi invia il messaggio non deve attendere la risposta
 - I messaggi vengono spesso numerati per meglio mostrare la sequenza temporale delle azioni

Sequence Diagram - Elementi

- **Messaggio**

- In UML 1.x i messaggi sincrónico ed asincroni erano visualizzati con una freccia con punta aperta completa o parziale,
- In UML 2.x per evitare facili confusioni si è preferito visualizzarli con una freccia chiusa piena ed una aperta completa mantenendo sempre la linea continua

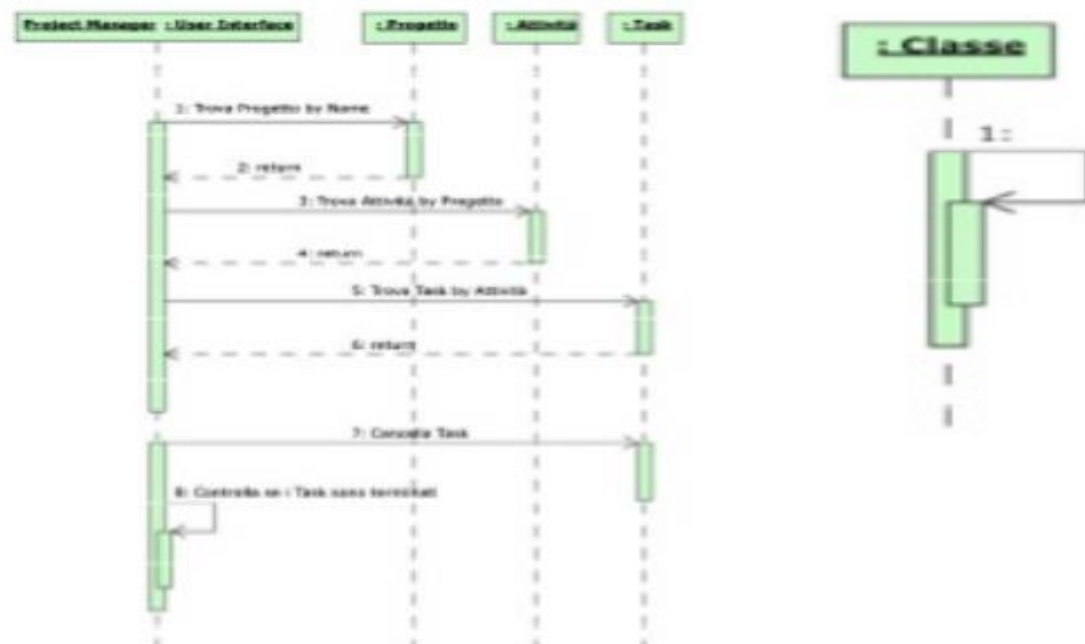


- Il messaggio può essere inviato anche a se stesso in modo ricorsivo

Sequence Diagram - Elementi

- **Messaggio Ricorsivo**

- Indica un messaggio inviato a se stesso
- E' utile quando si vogliono rappresentare comportamenti ricorsivi



Sequence Diagram - Elementi

- **Messaggio di ritorno**

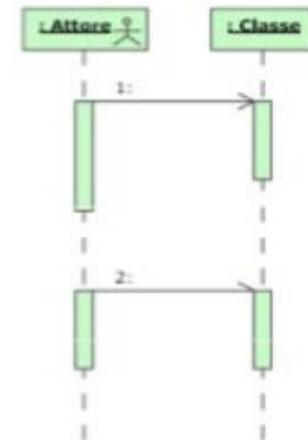
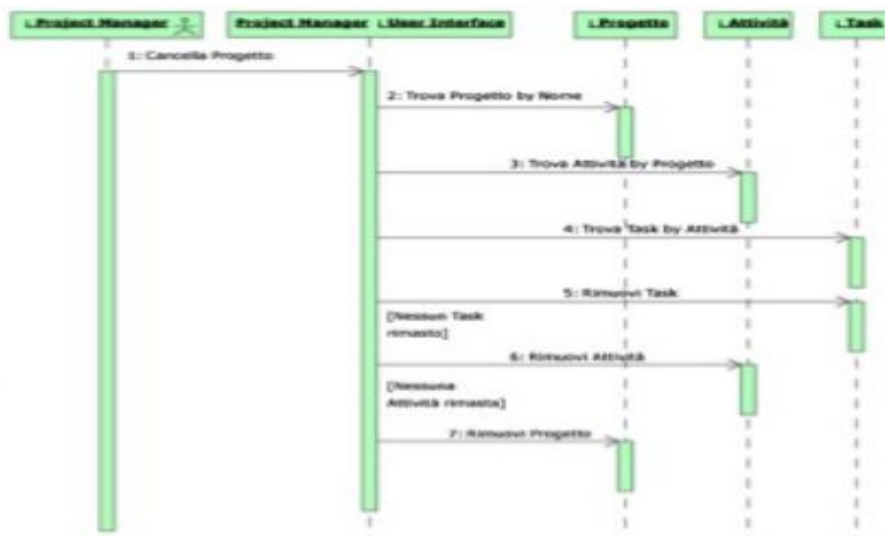
- Indica uno stimolo di ritorno dopo l'invio di un messaggio
- Non rappresenta un nuovo messaggio
- Si commette l'errore di utilizzarlo ogni volta che si invia un nuovo messaggio
- Deve essere utilizzato solamente per aggiungere informazioni e per aumentare la comprensione del sistema



Sequence Diagram - Elementi

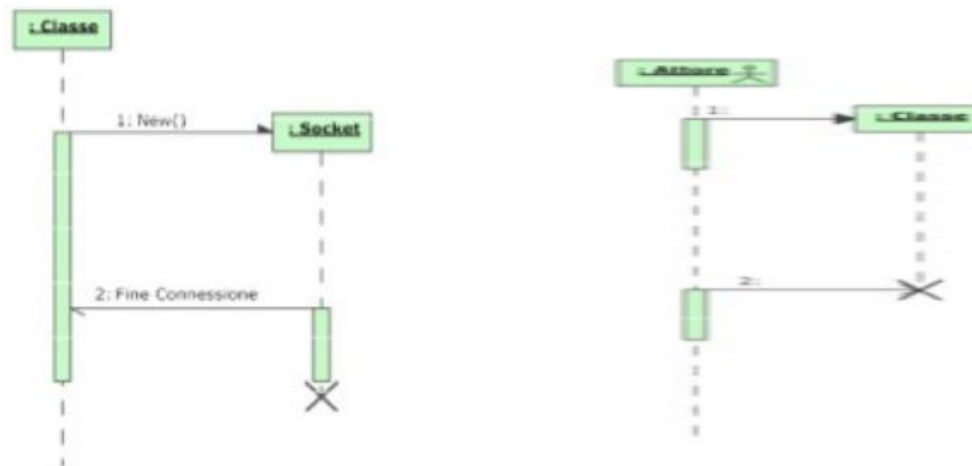
- **Attivazione**

- Rappresenta il tempo durante il quale un oggetto esegue un'operazione



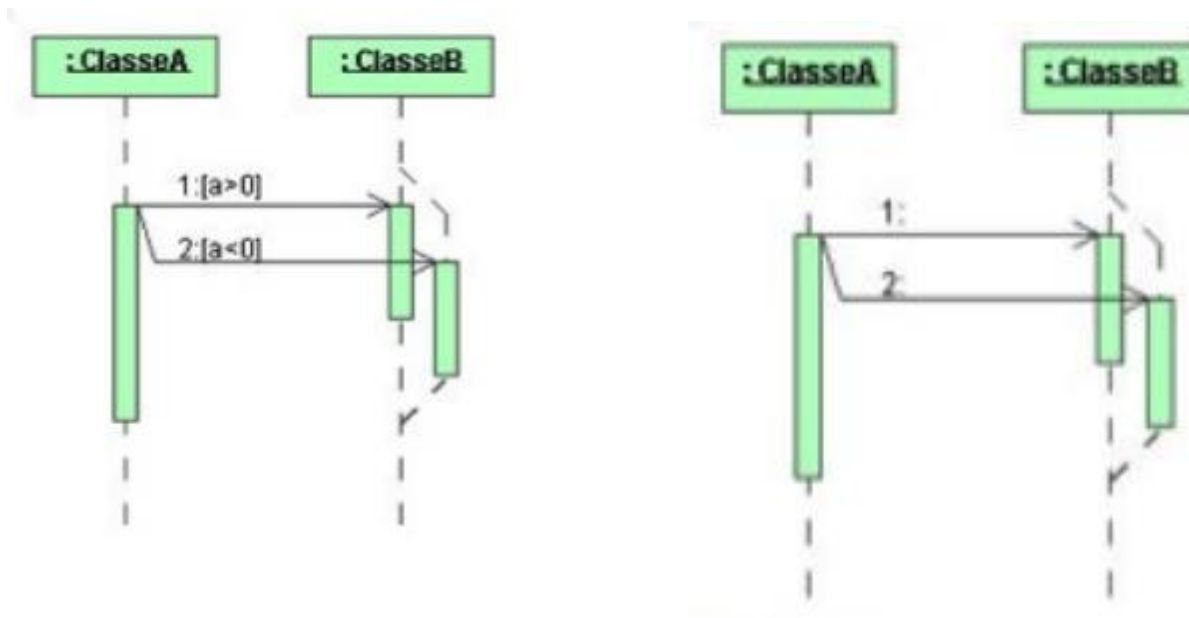
Sequence Diagram -Elementi

- **Creazione e distruzione**
 - La X indica quando una particolare istanza cessa di esistere
 - Un oggetto può morire da solo o grazie all'invio di un messaggio da parte di un altro oggetto
 - E' utile quando sto rappresentano contesti multithread



Sequence Diagram - Elementi

- **Esecuzione concorrente**
 - Indica quando alcune condizioni possono generare cammini diversi



Modulo 5

Packaging e documentazione

A series of horizontal lines in teal and light blue colors, with varying lengths and thicknesses, extending from the right side of the title area across the slide.

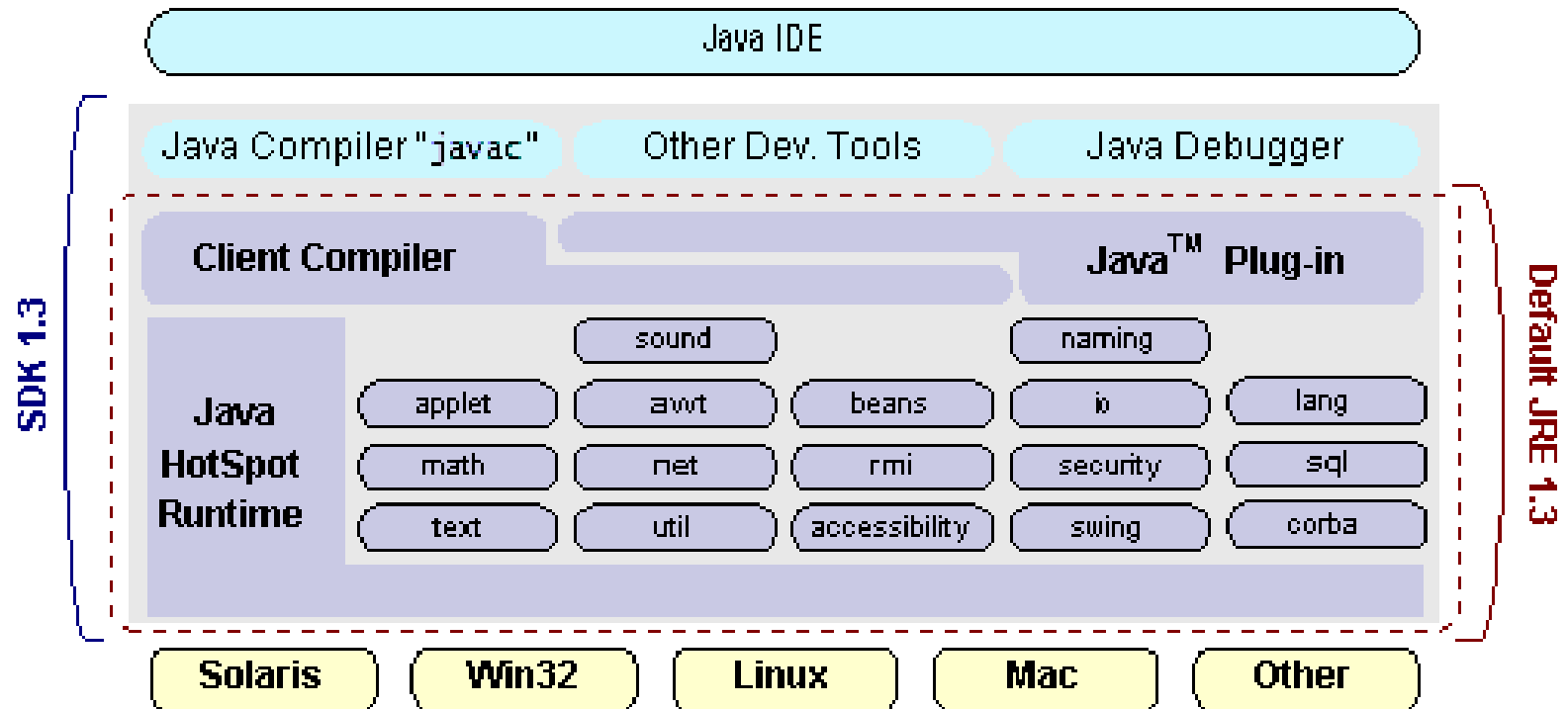
Lezione 1

I membri dei Package

- Le classi e le interfacce Java sono organizzate in package
- Rappresentano dei pacchetti in cui vengono inglobate le classi che presentano:
 - caratteristiche in comune (java.io)
 - appartengono allo stesso progetto (it.fastweb.crm.om.)
 - vengono prodotte dalla stessa società (com.cedati)

Lezione 1

I Package di sistema



Lezione 1

I membri dei Package

- I packages sono identificati dall'istruzione package.
- Essa deve apparire come la prima istruzione nel file del codice sorgente.
- Esempio:
 - ```
package mioPackage;
 public class MiaClasse {

 }
```

# Lezione 1

## Uso delle classi del Package

- Ci sono tre modi per utilizzare un membro (pubblico) definito in un package:
  1. attraverso il suo nome completo
  2. importandolo
  3. importando l'intero package a cui il membro appartiene

# Lezione 1

## Usando il nome completo

- Indichiamo:
  - il nome del package
  - l'operatore: che ci permette di entrare internamente nel package
  - il nome della classe

```
public class Classe {
 mioPackage. MiaClasse fg;
}
```

# Lezione 1

## Usando il nome della singola classe

- Indichiamo:
  - import seguito del nome del package seguito dalla classe
  - dichiariamo l'oggetto fg nel modo classico

```
import mioPackage.MiaClasse;

public class Classe {
 MiaClasse fg;

}
```

# Lezione 1

## Importando l'intero package

- Indichiamo:
  - import di tutte le classi del package attraverso il simbolo \*
  - dichiariamo l'oggetto fg

```
import mioPackage.*;

public class Classe {
 MiaClasse fg;

}
```

# Lezione 1

## Salvare e compilare una classe di un package

- Una classe appartenente ad un package deve essere salvata all'interno dell'albero che viene definito dal path del package:
- Esempio:
  - `package mioPackage.mioEsempio`
  - `class MiaClasse {...}`
- la classe `MiaClasse.java` deve essere salvata nelle cartelle `mioPackage\mioEsempio`
  - `c:\mioPackage\mioEsempio>dir`
  - `MiaClasse.java`
  - `c:\mioPackage\mioEsempio> javac MiaClasse.java`
  - `c:\mioPackage\mioEsempio>dir`
  - `MiaClasse.java`
  - `MiaClasse.class`

# Lezione 1

## Eseguire una classe del package

- Per eseguire una classe contenuta in un package occorre lanciare la classe (che contiene il main) dall'esterno della prima cartella che delimita il package
- Esempio:
  - `C:\mioPackage\mioEsempio>dir`
  - `MiaClasse.java`
  - `MiaClasse.class`
  - `c:\>java mioPackage.mioEsempio.MiaClasse`



# Lezione 2

## Import e Import statici

- Mentre l'istruzione import consente di importare le classi o interfacce, l'import statico consente di importare proprietà static di una classe:

```
import static java.lang.System.out;
public class ProvaImportStatic {
 public static void main(String[] args) {
 out.println("Hello");
 }
}
```

# Lezione 3

## Documentazione e javadoc

- E' possibile generare direttamente dai sorgenti java la documentazione dello sviluppo.
- I commenti al codice vengono inseriti nella documentazione per essere leggibili dal browser.
- Per fare questo si utilizza l'utility presente nel JDK dal nome "javadoc"

# Lezione 3

## Documentazione e javadoc

- Partiamo da una classe appositamente commentata:

```
/** Classe di Prova */
public class Prova {

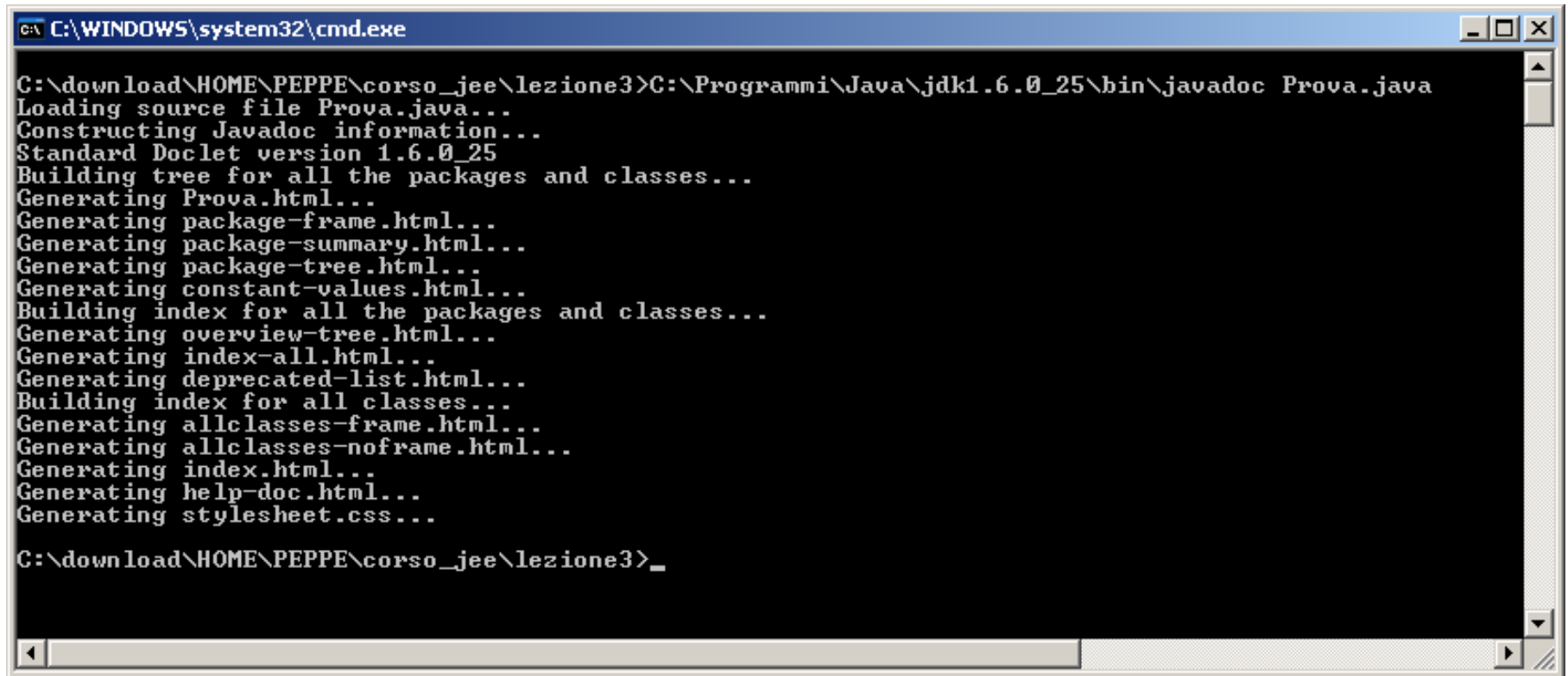
 /** Proprietà privata x */
 private int x = 0;

 /** Descrizione del metodo */
 public String metodo(int t) throws Exception {
 return "";
 }

}
```

# Lezione 3

## Documentazione e javadoc



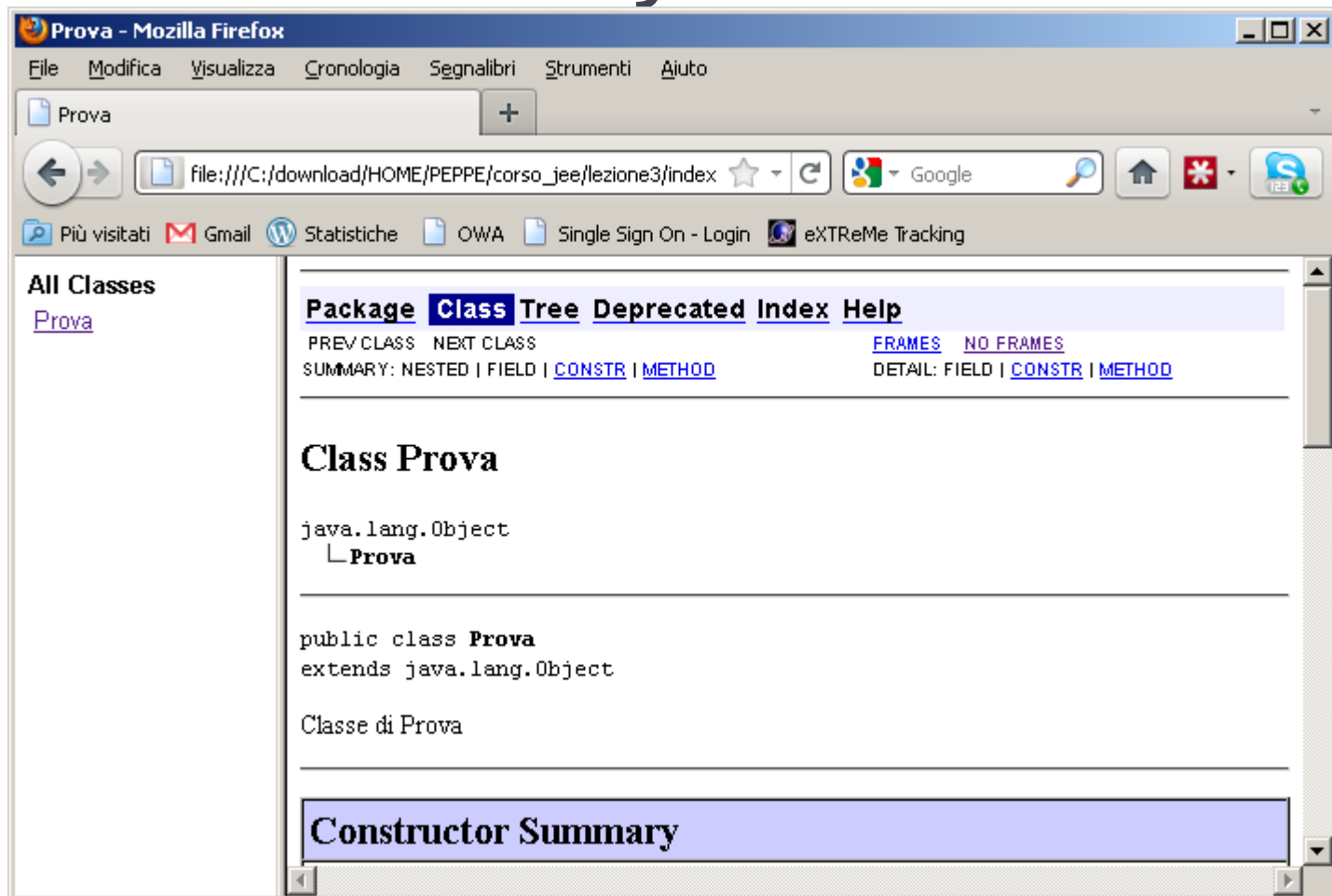
```
C:\WINDOWS\system32\cmd.exe

C:\download\HOME\PEPPE\corso_jee\lezione3>C:\Programmi\Java\jdk1.6.0_25\bin\javadoc Prova.java
Loading source file Prova.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_25
Building tree for all the packages and classes...
Generating Prova.html...
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating allclasses-noframe.html...
Generating index.html...
Generating help-doc.html...
Generating stylesheet.css...

C:\download\HOME\PEPPE\corso_jee\lezione3>_
```

# Lezione 3

## Documentazione e javadoc



# Lezione 4

## Annotazioni

- Le annotazioni, come dice il nome, vengono create per “annotare” degli aspetti importanti del programma in modo da rendere il codice più leggibile.
- Vengono usate spesso per passare delle note al compilatore in modo tale che questi possa svolgere il proprio compito in un modo più efficiente.

# Lezione 4

## Annotazioni

- Usiamo una “annotazione” nel codice:

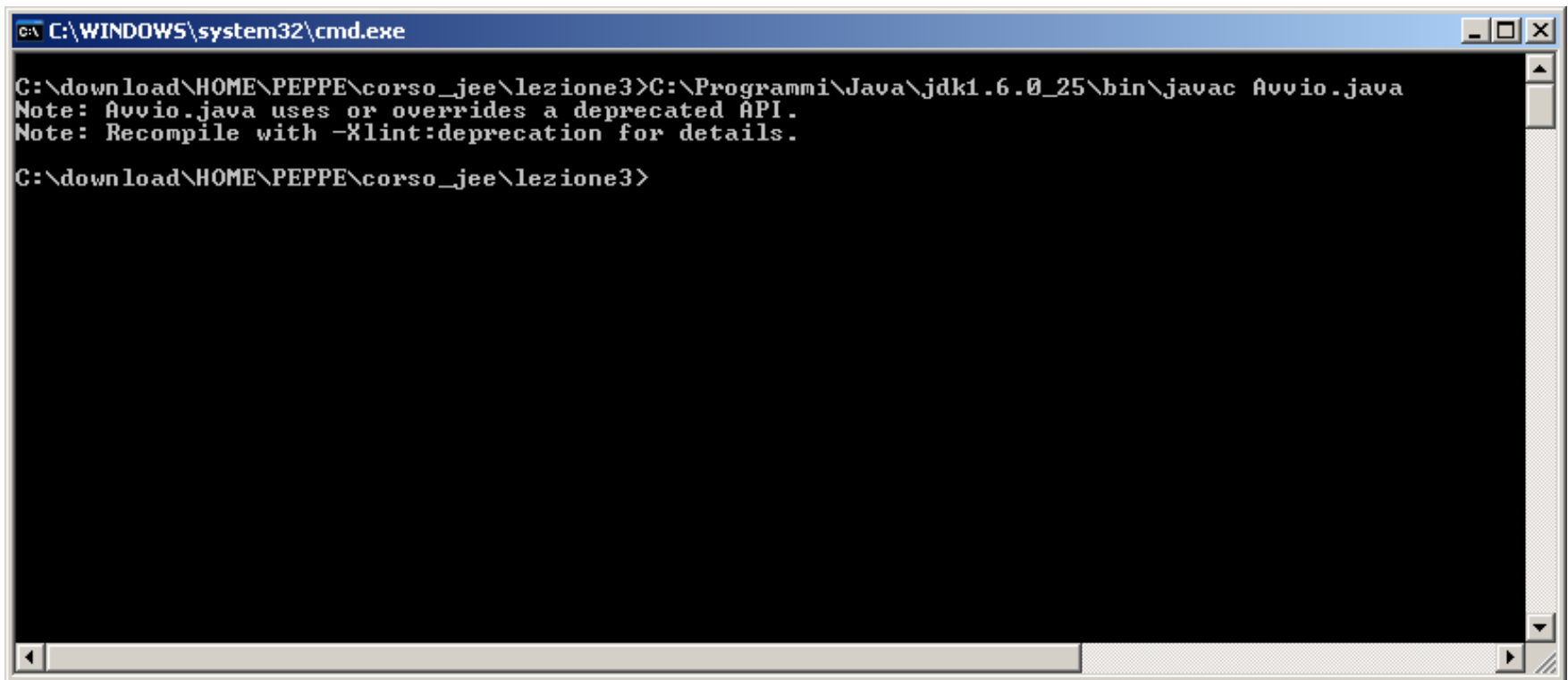
```
public class MiaProva {
 @Deprecated
 public void metodoConNotazione(){}
}
```

```
public class Avvio{
 public static void main(String [] args){
 new MiaProva().metodoConNotazione();
 }
}
```

# Lezione 4

## Annotazioni

- Abbiamo questo output:



```
C:\WINDOWS\system32\cmd.exe

C:\download\HOME\PEPPE\corso_jee\lezione3>C:\Programmi\Java\jdk1.6.0_25\bin\javac Avvio.java
Note: Avvio.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\download\HOME\PEPPE\corso_jee\lezione3>
```



# Lezione 4

## Annotazioni

- Creiamo una nostra “annotazione” ed usiamola nel codice:

```
public @interface ProvaAnnotazione {
 String nome();
}
```

```
public class MiaProva {
 @ProvaAnnotazione(nome="primaAnnotazione")
 public void metodoConNotazione(){}
}
```

# Modulo 6

## Librerie base della J2SE

A series of horizontal lines in teal and light blue colors, with varying lengths and thicknesses, extending from the left edge of the slide towards the right.

# Lezione 1

## Il package java.lang

- Il package java.lang è importato di default in quanto contiene la struttura di base per la creazione di una classe java.
- E' composto da:
  - interfacce: Cloneable, Comparable, Runnable
  - classi: Wrapper( Byte, Short, Integer, Long, Float, Double, Boolean, Character, Void ) String, System, Thread, Object, Runtime, SecurityManager,
  - eccezioni: Throwable

# Lezione 2

## Il package java.util

- Il package java.util contiene delle classi di utility che servono al programmatore per implementare i propri algoritmi.
- Per esempio: liste ordinate/casuali con valori unici/duplicati, tabelle di elementi, calendari, comparatori di elementi, gestori della localizzazione geografica
- interfacce/classi: List, Set, Map, Collection, Iterator, Enumeration, Calendar, Comparator, Local

# Lezione 3

## Il package java.text

- Il package java.text permette di convertire oggetti in testo e viceversa.
- Per esempio: DateFormat, DecimalFormat, MessageFormat, NumberFormat,

# Modulo 7

## Ereditarietà

A series of horizontal lines in teal and light blue colors, located on the right side of the slide, extending from the left edge of the white area.

# Lezione 1

## Boxing e Unboxing

- Viene utilizzato per assegnare ad un oggetto un tipo primitivo senza ricorrere a metodi di utility.
- Consente di velocizzare lo sviluppo di codice e di ottimizzare le prestazioni:

```
//senza boxing
Integer i1 = new Integer(100);
int y1 = 20;
int z1 = y1 + i1.intValue();
```

```
//con boxing
Integer i2 = 100;
int y2 = 20;
int z2 = y2 + i2;
```

# Lezione 2

## Concetti di ereditarietà

- L'ereditarietà consente di definire un tipo (=classe) sulla base dei dati e dei metodi di un tipo già definito, ereditandone operativamente sia i dati che i metodi.
- La classe di partenza è la "classe base" o "superclasse" o "classe padre"; mentre la classe che si crea è la "classe derivata" o la "sottoclasse" o la "classe figlia".
- Nella terminologia OOP si usa anche il termine "estendere" che indica la creazione di una classe derivata da una classe base.
- Non bisogna mai confondere l'ereditarietà con l'incorporamento (classe contenitore)!
- Di grosso aiuto sono, nel linguaggio corrente, i verbi "E' un" e "Ha un".



# Lezione 2

## Concetti di ereditarietà

- La classe Figlio eredita la proprietà x dalla classe Padre e può usarla al suo interno:

```
class Padre {
 int x = 5;
}

public class Figlio extends Padre {
 public static void main(String[] args) {
 Figlio pr = new Figlio();
 pr.leggo();
 }
 void leggi() {
 System.out.println(x);
 }
}
```

# Lezione 3

## Polimorfismo

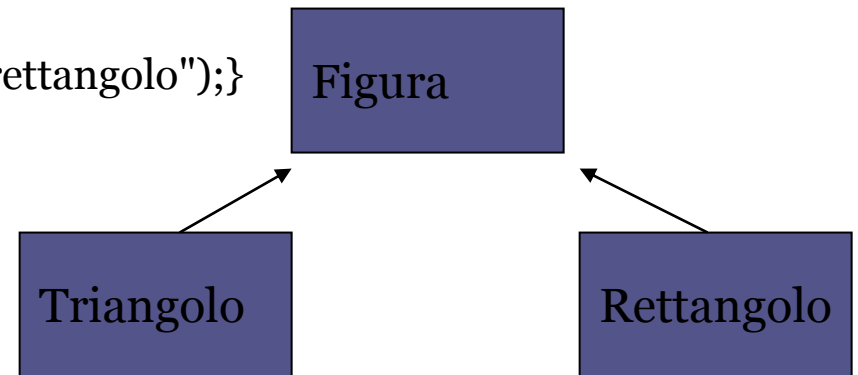
- Derivato dal greco, significa “pluralità di forme”
- E’ la caratteristica che ci consente di utilizzare un’unica interfaccia per una moltitudine di azioni. Quale sia la particolare azione eseguita dipende solamente dalla situazione in cui ci si trova.
- Mentre con gli oggetti il collegamento con la classe avviene a compile-time, nel caso del polimorfismo avviene a run-time
- Il polimorfismo viene impiegato impiegando una interfaccia, una classe astratta o una classe.
- E’ necessario che ci sia ereditarietà tra le classi

# Lezione 3

## Polimorfismo

Esempio:

```
abstract class Figura {
 abstract void rispondo();
}
class Triangolo extends Figura {
 void rispondo(){System.out.println("Sono il triangolo");}
}
class Rettangolo extends Figura {
 void rispondo(){System.out.println("Sono il rettangolo");}
}
```



# Lezione 3

## Polimorfismo

continuo...

```
public class Altro {
 Figura pr;
 public static void main(String[] args) {
 Altro al = new Altro();
 al.metodo();
 }
 void metodo() {
 pr = new Triangolo();
 pr.rispondo();
 pr = new Rettangolo();
 pr.rispondo();
 }
}
```

Tutto cio è possibile se esiste ereditarieta!!

# Lezione 4

## La classe Object

- La classe Object è la classe padre di tutte le classi. Se un classe non dichiara l'estensione, viene assegnata come estensione la classe Object.
- Tale classe consente di ereditare alcuni metodi e proprietà importanti come:
  - `hashCode()`, `equal()`, `length`

# Lezione 5

## L'overriding

- Con il termine overriding si intende la ridefinizione di un metodo ereditato.
- Una sottoclasse specializza il comportamento del padre e quindi spesso c'è la necessità di ridefinirne il codice, riscrivendolo o aggiungendo delle funzionalità.
- In alcuni casi è proprio necessario implementare dei metodi (vedi Classi astratte e Interfacce).

# Lezione 5

## L'overriding

- Esempio di overriding di un metodo:

```
class Padre {
 public void metodo() { codice }
}
class Figlio extends Padre {
 public void metodo() { nuovo codice }
}
```

# Lezione 5

## L'overriding

- Esempio di overriding con riscrittura del codice:

```
class Padre {
 int prop;
 public void metodo() { prop=5; }
}
class Figlio extends Padre {
 int prop2;
 public void metodo() { prop2=13; prop=14; }
}
```



# Lezione 5

## L'overriding

- Esempio di overriding con aggiunta di codice:

```
class Padre {
 int prop;
 public void metodo() { prop=5; }
}
class Figlio extends Padre {
 int prop2;
 public void metodo() { super(); prop2=3; }
}
```

# Lezione 6

## I generics

- I generic, introdotti con java 5, consentono di definire degli insiemi di oggetti vincolando la tipologia di oggetti:

```
//Senza Generic
Vector v = new Vector();
v.add("ciao");
v.add(new Integer(1));
```

```
//Con Generic
Vector<String> v2 = new Vector<String>();
v2.add("ciao");
v2.add(new Integer(1));
```

# Modulo 8

## Astrazioni

A series of horizontal lines in teal and light blue colors, some solid and some dashed, extending across the bottom of the slide.

# Lezione 1

## Le classi astratte

- Una classe astratta può definire metodi astratti, ossia metodi di cui non viene fornita l'implementazione e prevedere, in senso generico, quali comportamenti (metodi) dovranno caratterizzare le sue classi discendenti
- Le definizioni esatte dei comportamenti verranno però effettuate dalle classi eredi (istanziabili), cosicché possano riflettere le loro specifiche esigenze
- Una classe astratta può però contenere anche metodi non astratti

# Lezione 1

## Le classi astratte

```
abstract class MiaSuperClasseAstratta {
 void mioMetodoImplementato(...) {...implementazione};
 abstract void mioMetodo(...);
}
```

```
class MiaClasseErede extends MiaSuperClasseAstratta {
 void mioMetodo(...) {
 // implementazione
 }
}
```

- Occorre sempre implementare i metodi astratti nella classe erede altrimenti si genera un errore in compile-time

# Lezione 2

## Le interfacce

- Le interfacce
  1. ci forniscono una forma di eredità multipla
  2. ci permettono di dichiarare metodi di istanza e proprietà static final
  3. possiamo dichiarare solo metodi astratti
  4. non possiamo fissare livelli di accesso diversi da public
  5. non possiamo istanziare una interfaccia

# Modulo 9

## La gestione degli errori

A series of horizontal lines in teal and light blue colors, with varying lengths and slight offsets, creating a modern, layered effect across the width of the slide.

# Lezione 1

## Meccanismo vincolante delle eccezioni

- Una eccezione è un evento che si verifica durante l'esecuzione di un programma e che ne impedisce la normale prosecuzione a causa di errori hardware o semplici errori di programmazione.
- Il sistema runtime, per la gestione dell'eccezione, cerca "candidati" a partire dal metodo nel quale l'eccezione è stata "sollevata", proseguendo eventualmente lungo lo stack delle chiamate, finché non trova un metodo contenente un gestore(exception handler) opportuno.
- Se non viene trovato nessun exception handler adatto alla gestione di un'eccezione, il programma termina.



# Lezione 1

## Meccanismo vincolante delle eccezioni

Una eccezione potrà essere:

- gestita attraverso gli exception handler
- propagata lungo lo stack delle chiamate dei metodi

Un exception handler è composto da tre parti principali:

- un blocco try
- uno o più blocchi catch (opzionali)
- un blocco finally

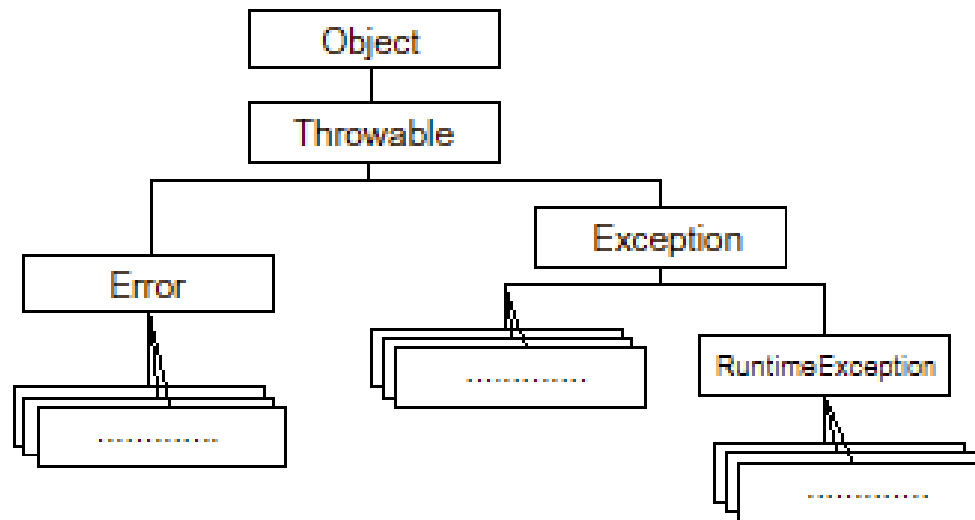
Nel caso si decida di propagare una eccezione lungo lo stack delle chiamate occorre utilizzare la parola chiave `throws` al metodo.

- La propagazione dell'eccezione consente ad un metodo di non doversi preoccupare di gestirlo.
- Se i metodi chiamanti non hanno previsto la gestione dell'errore, allora il programma si interromperà

# Lezione 2

## Le eccezioni di RunTime

- Alcune eccezioni possono verificarsi a causa di errori di coloro che utilizzano il programma e non ad opera di coloro che l'hanno realizzato.
- Possono sempre presentarsi ed il programmatore può semplicemente provare a gestirle ma non può impedirle.



# Lezione 3

## Eccezioni personalizzate

- La creazione di eccezioni da parte del programmatore nasce dalla necessità di gestire efficacemente situazioni di errore (o comunque anomale) non previste dalle eccezioni standard di Java ma che si possono comunque verificare durante l'esecuzione del programma che si sta progettando
- Esempio:

```
public class ElementoInesistenteException extends Exception {
 public String nomeElemento;
 public ElementoInesistenteException(String nome) {
 super("Non esiste alcun elemento" + nome);
 nomeElemento = nome;
 }
}
```