

Programmazione 3 e Laboratorio di Programmazione 3

Tipi di dati fondamentali

Proff. Angelo Ciaramella – Emanuel Di Nardo

Identificatori

- Java è un linguaggio **case sensitive**
- **Identificatori (nomi)**
 - **metodi, classi, oggetti, variabili, costanti, interfacce, enumerazioni, annotazioni**
- **Due regole**
 - **Non** può coincidere con una **parola chiave** (keyword) di Java
 - In un identificatore
 - il **primo carattere** può essere A-Z, a-z, _, \$
 - il **secondo ed i successivi** possono essere A-Z, a-z, _, \$, 0-9



Keywords

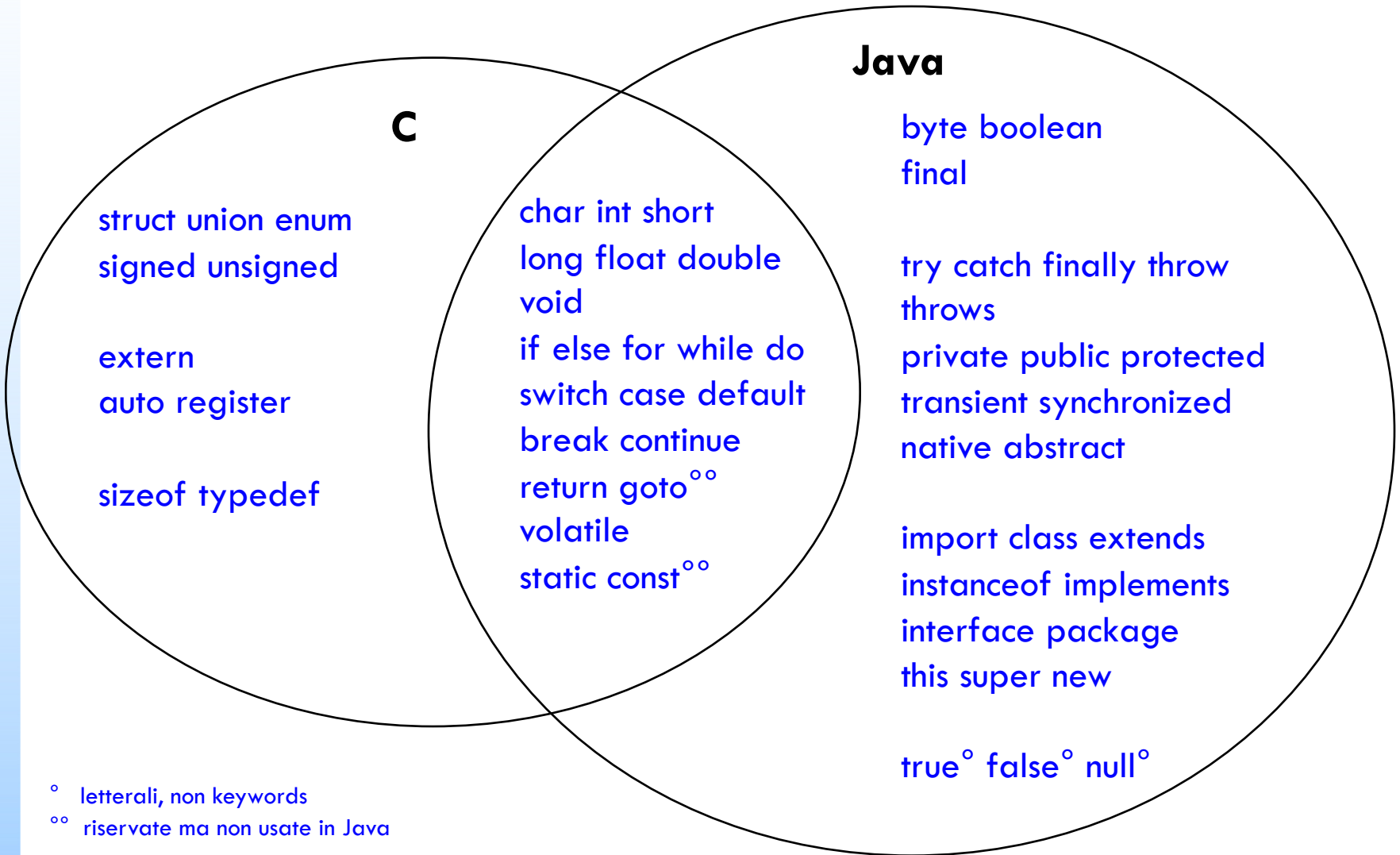
Le seguenti **keywords** non possono essere usate come identificatori

abstract	double	int	super
boolean	else	interface	switch
break	extends	long	synchronized
byte	final	native	this
case	finally	new	throw
catch	float	package	throws
char	for	private	transient
class	(goto)	protected try	
(const)	if	public	void
continue	implements	return	volatile
default	import	short	while
do	instanceof	static	

Note: - **const** e **goto** sono riservate, ma non usate
- anche i letterali **null**, **true**, **false** sono riservati



Keywords



Convezioni per i nomi

- Esistono **direttive** fornite direttamente da **Oracle** (originariamente da **Sun Microsystems**) per raggiungere uno standard anche nello **stile d'implementazione**
- **Identificatori**
 - devono essere **significativi**
 - di solito l'identificatore di una variabile è **composto da uno o più sostantivi** (`numeroLatI`)



Convezioni per i nomi - CamelCase

■ Classi

- Un identificatore di una classe (ma questa regola vale anche per le interfacce, le enumerazioni e le annotazioni che studieremo più avanti) deve sempre **iniziare con una lettera maiuscola** (`MacchinaDaCorsa`)

■ Variabili

- deve sempre iniziare con una **lettera minuscola**
- stesse regole viste per le classi (`pesoSpecifico`)



Convezioni per i nomi - CamelCase

■ Metodi

- stesso criterio per le variabili (`sommaDueNumeri(int a, int b)`)

■ Costanti

- tutte le lettere dovranno essere maiuscole (`PI_GRECO`)

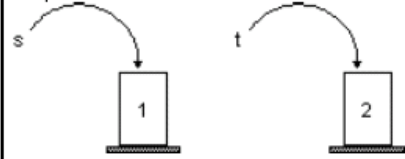
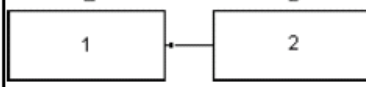
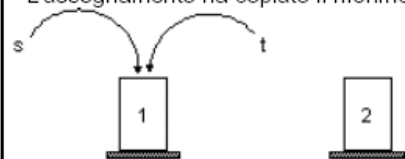

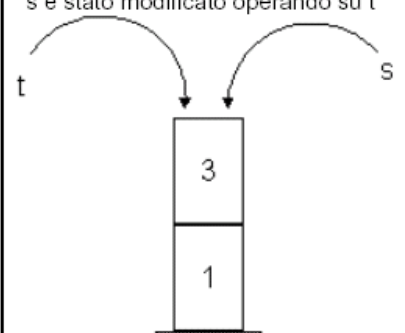


Tipi di dati

■ Ci sono due gruppi di tipi di dato

■ Primitivo

■ Oggetto

<pre>int a = 1;</pre> <p>Utilizzata una costante (1)</p>	<pre>Stack s=new Stack();</pre> <pre>Stack t= new Stack();</pre> <p>Utilizzato un <i>costruttore</i></p>
<pre>int b=a+1;</pre> <p>Operato con un operatore (+)</p>	<pre>s.push(1); t.push(2);</pre> <p>Operato con un metodo</p> 
<pre>a = b;</pre> <p>L'assegnamento ha copiato il valore</p> 	<pre>t = s;</pre> <p>L'assegnamento ha copiato il riferimento</p> 
<pre>a = 3;</pre> <p>b <i>non</i> è stato modificato</p> 	<pre>t.push(3);</pre> <p>s è stato modificato operando su t</p> 



Tipi di dati

	TIPI		KEYWORD	NOTE
Primitivi	boolean:		boolean	true, false
	numerici	interi	byte short int long char	8 bit interi in compl. a 2 16 bit 32 bit 64 bit 16 bit Unicode
		floating-point	float double	32 bit IEEE 754 64 bit
Reference	classi		class	
	interfacce		interface	
	array			
Null				



Intervalli numerici

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-2^{15}	$+2^{15}-1$	Short
int	32-bit	-2^{31}	$+2^{31}-1$	Integer
long	64-bit	-2^{63}	$+2^{63}-1$	Long
float	32-bit	IEEE754	IEEE754	Float
double	64-bit	IEEE754	IEEE754	Double
void	—	—	—	Void

Il pacchetto `java.math` permette di gestire numeri molto grandi, usando gli oggetti delle classi `BigInteger` e di `BigDecimal`; non hanno limite di dimensione e precisione, ma le operazioni sono rallentate



Tipizzazione

- **Statica**
 - Controllo a compile-time. Il tipo della variabile non può cambiare.
- **Forte**
 - Non avvengono conversioni automatiche
 - es. `String > int`
- **Nominativa**
 - Tipi con la stessa struttura devono esplicitare la loro relazione per essere considerati uguali
- **Manifest**
 - Il tipo deve essere sempre esplicitato (no type inference - jdk < [8, 10])



Dati interi

```
byte b = 10; //notazione decimale: b vale 10
short s = 022; //notazione ottale: s vale 18
long l = 0x12acd; //notazione esadecimale: l vale
76493
int i = 1000000000; //notazione decimale: i vale
1000000000
int n = 0b10100001010001011010000101000101
//notazione binaria:
//n vale -1589272251

int i = 1_000_000_000;
int n = 0b10100001_01000101_10100001_01000101

byte b = (byte) 257;
```

cast

Migliorare la leggibilità



Virgola mobile

```
double d = 1.26E-2;  
//equivalente a 1.26 diviso 100 = 0.0126
```

```
float f = 3.14F;
```

```
double d = 10.12E24D;
```



Letterale

```
char primoCarattere = 'a';
char car = '@';
char letteraGrecaPhi = '\u03A6'; //(lettera "Φ")
char carattereUnicodeNonIdentificato = '\uABC8';

/*
\n che equivale ad andare a capo (tasto new line)
\\ che equivale ad un solo \ (tasto backslash)
\t che equivale ad una tabulazione (tasto TAB)
\' che equivale ad un apice singolo
\" che equivale ad un doppio apice (virgolette)
*/
```



Promotion

- **Promozione automatica** nelle espressioni (promotion)
- Per gli operatori binari esistono **quattro regole** che dipendono dai tipi degli operandi in questione
 - se **uno degli operandi è double**, l'altro operando sarà convertito in **double**;
 - se il più ampio degli operandi è un **float**, l'altro operando sarà convertito in **float**;
 - se il più ampio degli operandi è un **long**, l'altro operando sarà convertito in **long**;
 - in ogni altro caso entrambi gli operandi saranno convertiti in **int**



Inizializzazione

- Ogni **variabile** in Java richiede che al momento della dichiarazione le venga assegnato un **valore iniziale**

```
identificatore var_name = var_value;
```

- Alternativamente, di **default**

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



Tipi di dati non primitivi

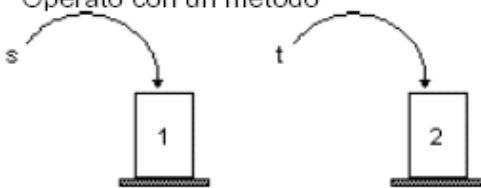
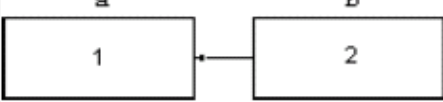
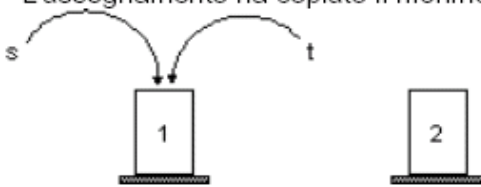
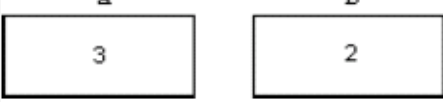
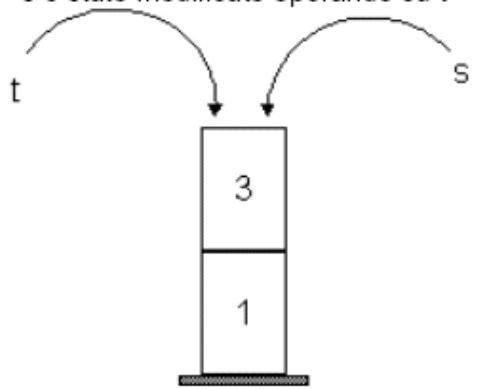
- Dichiariamo un **oggetto**

```
NomeClasse nomeOggetto;
```

- Il nome che diamo ad un oggetto è detto **reference**
 - una variabile *puntatore*
- Il **passaggio di parametri** in Java avviene sempre *per valore*
 - *pass-by-value*
 - *Attenzione all'utilizzo degli oggetti (mutable vs immutable)*



Primitivi vs dati non primitivi

<pre>int a = 1;</pre> <p>Utilizzata una costante (1)</p>	<pre>Stack s=new Stack();</pre> <pre>Stack t= new Stack();</pre> <p>Utilizzato un <i>costruttore</i></p>
<pre>int b=a+1;</pre> <p>Operato con un operatore (+)</p>	<pre>s.push(1); t.push(2);</pre> <p>Operato con un metodo</p> 
<pre>a = b;</pre> <p>L'assegnamento ha copiato il valore</p> 	<pre>t = s;</pre> <p>L'assegnamento ha copiato il riferimento</p> 
<pre>a = 3;</pre> <p>b <i>non</i> è stato modificato</p> 	<pre>t.push(3);</pre> <p>s è stato modificato operando su t</p> 

Classe String

- In Java le **stringhe** non sono array di caratteri (**char**), bensì oggetti
- Per assegnare un **valore** ad una stringa bisogna che esso sia **compreso tra virgolette (doppio apice ")**
 - a differenza dei **caratteri** per cui vengono utilizzati gli **apici singoli**



Stringhe

```
String nome = "Mario Rossi";  
// oppure  
String nome = new String("Mario Rossi");  
  
String a = "giorgio";  
String b = a.toUpperCase();  
System.out.println(a); // a rimane immutato  
  
System.out.println(b); // b è la stringa maiuscola  
  
/* produrrebbero il seguente output:  
giorgio  
GIORGIO  
*/
```



Operatori

- Operatori
 - Java eredita in blocco tutti gli operatori del linguaggio C



Operatori

Elenco degli **operatori** Java ordinati secondo l'**ordine di precedenza**; dal più alto al più basso.

Operatori	Funzioni
++ -- + - ?	Aritmetiche unarie e booleane
* / %	Aritmetiche
+ -	Addizione, sottrazione e concatenazione
<< >> >>>	Shift di bit
< <= > >= <i>instanceof</i>	Comparazione
== !=	Uguaglianza e disuguaglianza
&	(bit a bit) AND
^	(bit a bit) XOR
	(bit a bit) OR
&&	AND Logico
	OR Logico
!	NOT Logico
expr ? expr :expr	Condizione a tre
= *= /+ %= += -= <<= >>= n &= ^= =	Assegnamento e di combinazione

Operatori

- La differenza con C sta nel fatto che gli **operatori logici** in Java sono di tipo “**short-circuit**”
 - Se il **lato sinistro** di una espressione fornisce informazioni sufficienti a completare l'intera operazione, il **lato destro** della espressione non verrà valutato

```
if (false && true) { ... }
```

- Nell'esempio l'elemento di sinistra è false e dato che c'è l'operatore && è inutile valutare anche ciò che c'è alla sua destra



Operatori unari

```
i++      // equivale a i=i+1
++i

i--      // equivale a i=i-1
--i

i*=2     // equivale a i=i*2
```

Forma shortcut	Forma estesa corrispondente	Risultato dopo l'esecuzione
<pre>int i=0; int j; j=i++;</pre>	<pre>int i=0; int j; j=i; i=i+1;</pre>	<pre>j=1 j=0</pre>
<pre>int i=1; int j; j=i--;</pre>	<pre>int i=1; int j; j=i; i=i-1;</pre>	<pre>j=0 j=1</pre>
<pre>int i=0; int j; j=++i;</pre>	<pre>int i=0; int j; i=i+1; j=i;</pre>	<pre>j=1 j=1</pre>
<pre>int i=1; int j; j=--i;</pre>	<pre>int i=1; int j; i=i-1; j=i;</pre>	<pre>j=0 j=0</pre>



Operatori aritmetici

Operatori Aritmetici Binari		
Operatore	Utilizzo	Descrizione
+	res=sx + dx	res = somma algebrica di dx ed sx
-	res= sx - dx	res = sottrazione algebrica di dx da sx
*	res= sx * dx	res = moltiplicazione algebrica tra sx e dx
/	res= sx / dx	res = divisione algebrica di sx con dx
%	res= sx % dx	res = resto della divisione tra sx e dx



Operatori relazionali

Operatori Relazionali		
Operatore	Utilizzo	Descrizione
>	res=sx > dx	res = true se e solo se sx è maggiore di dx
>=	res= sx >= dx	res = true se e solo se sx è maggiore o uguale di dx
<	res= sx < dx	res = true se e solo se sx è minore di dx
<=	res= sx <= dx	res = true se e solo se sx è minore o uguale di dx
!=	res= sx != dx	res = true se e solo se sx è diverso da dx

L'operatore == verifica l'uguaglianza



Confrontare stringhe

- Per verificare se due stringhe sono uguali fra loro si usa

```
if (string1.equals(string2))
```

- L'operatore `==` verifica se si riferiscono allo stesso oggetto



Confrontare stringhe

- **Confronto di stringhe** secondo l'ordine alfabetico

```
string1.compareTo(string2) == 0  
//verifica se sono uguali
```

```
string1.compareTo(string2) < 0  
//string1 precede nell'ordine alfabetico
```

```
string1.compareTo(string2) > 0  
//string2 precede string1 nell'ordine alfabetico
```



Confrontare oggetti

- `==` verifica se due riferimenti a oggetto sono identici
- Per confrontare i contenuti di oggetti si usa `equals`
 - Il metodo deve essere dichiarato nelle classi
 - Ad esempio `Rectangle` ha un metodo `equals`

Codice di riferimento

```
EqualsMethod.java, EqualsMethod2.java,  
Equivalence.java
```



Operatori condizionali

Operatori Condizionali		
Operatore	Utilizzo	Descrizione
&&	res=sx && dx	AND : res = true se e solo se sx vale true e dx vale true, false altrimenti.
	res= sx dx	OR : res = true se e solo se almeno uno tra sx e dx vale true, false altrimenti.
!	res= ! sx	NOT : res = true se e solo se sx vale false, false altrimenti.
^	res= sx ^ dx	XOR : res = true se e solo se uno solo dei due operandi vale true, false altrimenti.



Operatori di shift bit a bit		
Operatore	Utilizzo	Descrizione
>>	<code>sx >> dx</code>	Sposta i bit di <code>sx</code> verso destra di un numero di posizioni come stabilito da <code>dx</code> .
<<	<code>sx << dx</code>	Sposta i bit di <code>sx</code> verso sinistra di un numero di posizioni come stabilito da <code>dx</code> .
>>>	<code>sx >>> dx</code>	Sposta i bit di <code>sx</code> verso sinistra di un numero di posizioni come stabilito da <code>dx</code> , ove <code>dx</code> è da considerarsi un intero senza segno.

Bitwise

Operatori logici bit a bit		
Operatore	Utilizzo	Descrizione
&	res = sx & dx	AND bit a bit
	res = sx dx	OR bit a bit
^	res = sx ^ dx	XOR bit a bit
~	res = ~sx	COMPLEMENTO A UNO bit a bit

AND (&)		
sx (bit)	dx (bit)	res (bit)
1	1	1
1	0	0
0	1	0
0	0	0

COMPLEMENTO (~)	
sx (bit)	res (bit)
1	0
0	1

OR ()		
sx (bit)	dx (bit)	res (bit)
1	1	1
1	0	1
0	1	1
0	0	0

XOR (^)		
sx (bit)	dx (bit)	res (bit)
1	1	0
1	0	1
0	1	1
0	0	0

Assegnamento

Operatori di assegnamento shortcut		
Operatore	Utilizzo	Equivalente a
+=	<code>sx +=dx</code>	<code>sx = sx + dx;</code>
-=	<code>sx -=dx</code>	<code>sx = sx - dx;</code>
*=	<code>sx *=dx</code>	<code>sx = sx * dx;</code>
/=	<code>sx /=dx</code>	<code>sx = sx / dx;</code>
%=	<code>sx %=dx</code>	<code>sx = sx % dx;</code>
&=	<code>sx &=dx</code>	<code>sx = sx & dx;</code>
=	<code>sx =dx</code>	<code>sx = sx dx;</code>
^=	<code>sx ^=dx</code>	<code>sx = sx ^ dx;</code>
<<=	<code>sx <<=dx</code>	<code>sx = sx << dx;</code>
>>=	<code>sx >>=dx</code>	<code>sx = sx >> dx;</code>
>>>=	<code>sx >>>=dx</code>	<code>sx = sx >>> dx;</code>

Codice di riferimento

`AllOps.java`, `AutoInc.java`, `Bool.java`, `Literals.java`,
`MathOps.java`, `URShift.java`



String

```
String a = "Java";  
String b = "Java";  
String c = new String("Java");  
System.out.println(a==b);  
System.out.println(b==c);
```

// produrrà il seguente output:

```
true  
false
```

```
System.out.println(a.equals(b));  
System.out.println(b.equals(c));  
// produrrà il seguente output:
```

```
true  
true
```



Costanti

- Una variabile `final` è una costante

```
public double getTotal()  
{  
  
    final double NICKEL_VALUE    = 0.05;  
    final double DIME_VALUE      = 0.1;  
    final double QUARTER_VALUE   = 0.25;  
  
    ...  
}
```



Costanti static

- Per poter utilizzare la costante in più metodi dichiariamole costanti insieme alle variabili istanza come `static final`

```
// Costanti utilizzabili da tutti i metodi  
  
private static final double NICKEL_VALUE    = 0.05;  
private static final double DIME_VALUE     = 0.1;  
private static final double QUARTER_VALUE  = 0.25;
```



Costanti static

- Spesso le **costanti** statiche si dichiarano **pubbliche**
 - Insieme di di *costanti per l'intera classe*

```
public double BankAccount
{
public static final double OVERDRAFT_FEE = 5;
...
}

// In qualsiasi metodo ci possiamo riferire a
BankAccount.OVERDRAFT_FEE;
```



Riassumendo...

- In un metodo

```
final nomeTipo nomeVariabile = espressione;
```

- In una classe

```
soecificatoreDiAccesso static final nomeTipo  
nomeVariabile = espressione;
```



Esercizio

- Implementare una classe `Purse` con i seguenti metodi

- `addNickels`
- `addDimes`
- `addQuarter`
- `getTotal`

- Il metodo `getTotal` calcola

```
nickels * 0.05 + dimes * 0.1 + quarters * 0.25
```

Codice di riferimento

```
Purse.java, PurseTest.java
```



PurseTest

```
public class PurseTest
{
    public static void main(String[] args)
    {
        Purse myPurse = new Purse();
        myPurse.addNickels(3);
        myPurse.addDimes(1);
        myPurse.addQuarters(2);
        double totalValue = myPurse.getTotal();
        System.out.print(" The total is ");
        System.out.print(totalValue);
    }
}
```



Metodi statici

- Un metodo statico (`static`) non agisce su nessun oggetto
 - Ad esempio `Math.sqrt`
- `Math` è una classe non un oggetto
 - I metodi statici sono sempre definiti all'interno delle classi
- Sintassi

```
NomeClasse.nomeMetodi (parametri) ;
```



Classe Math

- `Math.sqrt(x)`
- `Math.pow(x,y)`
- `Math.sin(x)`
- `Math.cos(x)`
- `Math.tan(x)`
- `Math.asin(x)`
- `Math.acos(x)`
- `Math.atan(x)`
- `Math.atan2(x,y)`
- `Math.exp(x)`
- `Math.log(x)`
- `Math.round(x)`
- `Math.ceil(x)`
- `Math.floor(x)`
- `Math.abs(x)`



Valutazione espressione

$$\begin{aligned} & (-b + \text{Math.sqrt}(b * b - 4 * a * c)) / (2 * a) \\ & \quad \underbrace{\quad \quad \quad}_{b^2} \quad \underbrace{\quad \quad \quad}_{4ac} \quad \underbrace{\quad \quad \quad}_{2a} \\ & \quad \quad \quad \underbrace{\quad \quad \quad}_{b^2 - 4ac} \\ & \quad \quad \quad \underbrace{\quad \quad \quad}_{\sqrt{b^2 - 4ac}} \\ & \quad \quad \quad \underbrace{\quad \quad \quad}_{-b + \sqrt{b^2 - 4ac}} \\ & \quad \quad \quad \underbrace{\quad \quad \quad}_{\frac{-b + \sqrt{b^2 - 4ac}}{2a}} \end{aligned}$$



Conversione di tipi

- Quando si converte un numero in virgola mobile in un intero abbiamo una **perdita di informazione**
- Per convertire un valore in un tipo diverso si usa un **cast**

```
int dollars = 2;  
double total = dollars;  
  
int dollars = (int) total;
```



Stringhe

- Una **stringa** è una sequenza di caratteri racchiusa tra virgolette “”
 - Oggetti della classe **String**
- Esempi

```
int n = message.length(); //Lunghezza stringa

// Concatenazione
String a = "Agent";
int n = 7;
String bond = a + n;

System.out.println("the total is " + total);
```



Stringhe

- Se una stringa contiene le **cifre di un numero** usiamo i metodi

```
Integer.parseInt  
Double.parseDouble
```

- Caratteri tutti maiuscoli o tutti minuscoli

```
String greeting = "Hello";  
System.out.println(greeting.toUpperCase());  
System.out.println(greeting.toLowerCase());
```

- Per estrarre una parte di una stringa si usa il metodo

```
s.substring(start, pastEnd);
```



Finestra di dialogo

- Finestra di dialogo che può ricevere dati

```
String input = JOptionPane.showInputDialog("xxxxx");
```

- Restituisce la stringa inserita dall'utente



Esempio

```
import javax.swing.JOptionPane;

...

Purse myPurse = new Purse();
String input = JOptionPane.showInputDialog("how many
nickels do you have?");
int count = Integer.parseInt(input);
myPurse.addNickels(count);

...
```

Codice di riferimento

[InputTest.java](#)



Input da console

- I dati di ingresso da console vengono letti dall'oggetto `System.in` (legge solo byte)
- Per ottenere un lettore di caratteri si deve trasformare `System.in` in un oggetto di tipo `InputStreamReader` (carattere alla volta)

```
InputStreamReader reader = InputStreamReader(System.in);
```



Input da console

- Possiamo leggere una stringa

```
BufferedReader console = new BufferedReader(reader) ;  
  
String input = console.readLine() ;  
int count = Integer.parseInt(input) ;
```

- Per gestire l'eccezione

```
public static void main(String[] args) throws  
IOException  
  
        oppure  
  
public void readInput(BufferedReader reader) throws  
IOException
```



Input da console - ConsoleReader

- La classe `ConsoleReader.java` permette di leggere da tastiera
- Testare la classe `Richter.java` per la classificazione dei terremoti

Codice di riferimento

`ConsoleReader.java`, `Richter.java`

