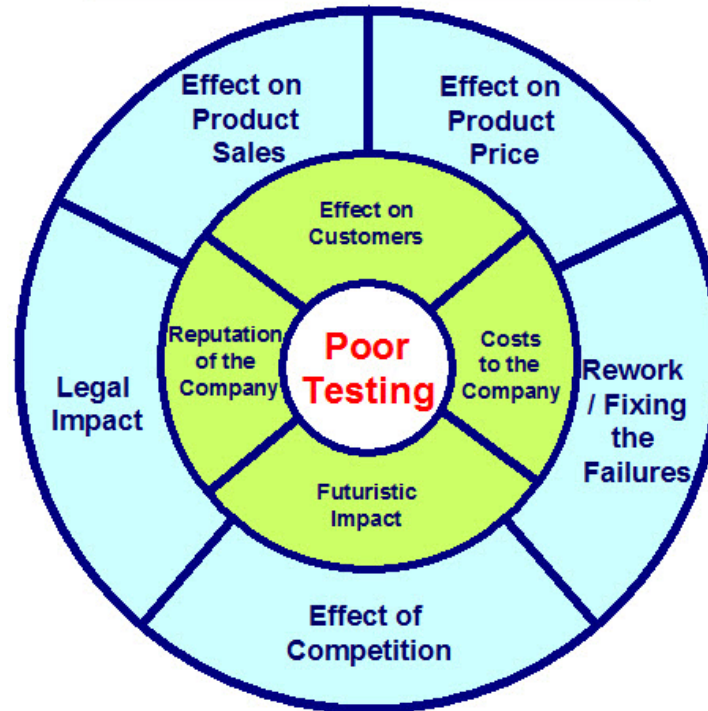


Collaudare una classe

Consequences of Poor Testing





Programmi di collaudo

- Usati per “collaudare” il funzionamento di una classe
- Passi per costruire un programma di collaudo
 - Definire una nuova classe
 - Definire in essa il metodo main
 - Costruire oggetti all'interno di main
 - Applicare metodi agli oggetti
 - Visualizzare risultati delle invocazioni dei metodi
- **ATTENZIONE:** bisogna *importare* le classi utilizzate

I pacchetti di classi (package)

- Tutte le classi della libreria standard sono raccolte in ***pacchetti*** (***package***) e sono organizzate per argomento e/o per finalità
 - *Esempio*: la classe **Rectangle** appartiene al pacchetto **java.awt** (Abstract Window Toolkit)
- Per ***usare*** una classe di una libreria, bisogna ***importarla*** nel programma, usando l'enunciato
 - **import** *nomePacchetto.NomeClasse*;
- Le classi **System** e **String** appartengono al pacchetto **java.lang**
 - il pacchetto **java.lang** viene ***importato automaticamente***



Esempio: MoveTester.java

```
import java.awt.Rectangle;
public class MoveTester
{
    public static void main(String[] args)
    {
        Rectangle box = new Rectangle(5, 10, 20, 30);

        // sposta il rettangolo
        box.translate(15, 25);

        // visualizza informaz. su rettangolo traslato
        System.out.println("After moving,
                           the top-left corner is:");
        System.out.println(box.getX());
        System.out.println(box.getY());
    }
}
```

È tutto chiaro? ...

- La classe Random è definita nel pacchetto java.util. Cosa bisogna fare per utilizzarla in un programma?
- Perché il programma MoveTester non visualizza altezza e larghezza del rettangolo dopo l'invocazione del metodo translate?

Tipi di dati fondamentali

- In java ci sono 8 **tipi di dati fondamentali** (o tipi di **dati primitivi**)
- Di questi, sei sono **tipi numerici**, quattro per numeri interi e due per numeri in virgola mobile

Tipo	Descrizione	Dimensione
int	Tipo intero con intervallo $-2147483648 \dots 2147483647$ (circa 2 miliardi)	4 byte
byte	Tipo che descrive un singolo byte, con intervallo $-128 \dots 127$	1 byte
short	Tipo intero “corto”, con intervallo $-32768 \dots 32767$	2 byte
long	Tipo intero “lungo”, con intervallo $-9223372036854775808 \dots 9223372036854775807$	8 byte
double	Tipo in virgola mobile a doppia precisione, con intervallo circa $\pm 10^{308}$ e circa 15 cifre decimali significative	8 byte
float	Tipo in virgola mobile a singola precisione, con intervallo circa $\pm 10^{38}$ e circa 7 cifre decimali significative	4 byte
char	Tipo che rappresenta caratteri codificati secondo lo schema Unicode (Argomenti avanzati 4.5)	2 byte
boolean	Tipo per i due valori logici true e false (Capitolo 6)	1 bit

Tipi di dati fondamentali

- Se servono i valori massimi/minimi dei numeri rappresentati con i vari tipi di dati non occorre ricordarli
- il pacchetto **java.lang** della libreria standard contiene una classe per ciascun tipo di dati fondamentali, in cui sono definiti questi valori come costanti

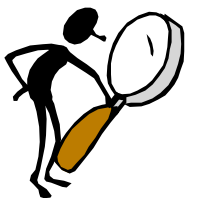
byte	Byte.MIN_VALUE	Byte.MAX_VALUE
short	Short.MIN_VALUE	Short.MAX_VALUE
int	Integer.MIN_VALUE	Integer.MAX_VALUE
long	Long.MIN_VALUE	Long.MAX_VALUE
float	Float.MIN_VALUE	Float.MAX_VALUE
double	Double.MIN_VALUE	Double.MAX_VALUE

Numeri interi in Java

- In Java tutti i tipi di dati fondamentali per numeri interi usano internamente la rappresentazione in complemento a due
- La JVM ***non segnala le condizioni di overflow*** nelle operazioni aritmetiche
 - **si ottiene semplicemente un risultato errato**
- L'unica operazione aritmetica tra numeri interi che genera una **eccezione** è la divisione con divisore zero
 - **ArithmeticException**



Intervalli numerici e precisione



- Come molte rappresentazioni di dati in un computer, anche **la rappresentazione di numeri** in Java **soffre di alcune limitazioni**, dovute a scelte progettuali di compromesso tra la precisione della rappresentazione e la velocità di elaborazione delle operazioni aritmetiche in Java
- I valori di tipo **int** sono compresi tra
 - **-2147483648** (costante **Integer.MIN_VALUE**)
 - **+2147483647** (costante **Integer.MAX_VALUE**)
- quindi, per esempio, la popolazione mondiale non può essere rappresentata con una variabile **int**

Altri tipi di dati interi

- Quando il tipo **int** non soddisfa le esigenze numeriche del problema, si possono usare altri tipi di dati interi
- Intervallo di variabilità insufficiente: tipo **long**
 - massimo valore assoluto con una variabile long: circa **9** miliardi di miliardi (**Long.MAX_VALUE** e **MIN_VALUE**)
 - per assegnare un valore a una variabile **long** bisogna aggiungere un carattere **L** alla fine

```
long x = 30000000000L;
```

- Esistono altri due tipi di dati per numeri interi
 - **byte**, con valori tra -128 e +127
 - **short**, con valori tra -32768 e +32767
- Si usano più raramente
 - alcuni metodi di classi della libreria standard richiedono l'uso di parametri di questo tipo



Numeri in virgola mobile in Java

- In Java i tipi di dati fondamentali per numeri in virgola mobile usano internamente una rappresentazione binaria codificata dallo standard IEEE 754
 - **float** (32bit), **double** (64 bit)
 - per assegnare un valore ad una variabile **float** bisogna aggiungere un carattere **f** alla fine

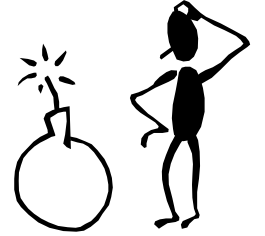
```
float x = 30.2f;
```

- La divisione con divisore zero non è un errore se effettuata tra numeri in virgola mobile
 - se il dividendo è diverso da zero, il risultato è **infinito** (con il segno del dividendo)
 - se anche il dividendo è zero, il risultato è indeterminato, cioè non è un numero, e viene usata la codifica speciale **NaN** (**Not a Number**)

Numeri in virgola mobile in Java

- Lo standard IEEE 754 prevede quindi anche la rappresentazione di NaN, di $+\infty$ e di $-\infty$
- Sono definite le seguenti costanti
 - **Double.NaN**
 - **Double.NEGATIVE_INFINITY**
 - **Double.POSITIVE_INFINITY**
- e le corrispondenti costanti **Float**
 - **Float.NaN**
 - **Float.NEGATIVE_INFINITY**
 - **Float.POSITIVE_INFINITY**

Errori di arrotondamento



- Gli errori di arrotondamento sono un fenomeno naturale nel calcolo in virgola mobile eseguito con un numero **finito** di cifre significative
 - calcolando $1/3$ con due cifre significative, si ottiene 0,33
 - moltiplicando 0.33 per 3, si ottiene 0.99 e non 1
- Siamo abituati a valutare questi errori pensando alla rappresentazione dei numeri in base **decimale**, ma i computer rappresentano i numeri in virgola mobile in base **binaria** e a volte si ottengono dei risultati inattesi!

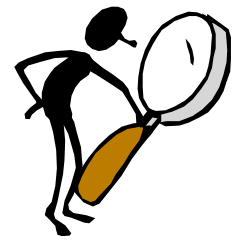
Arrotondamento

```
double f = 4.35F;  
System.out.println(100 * f);
```

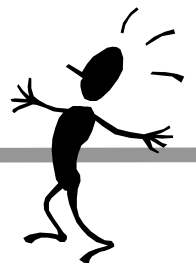
Stampa **434.999999999999999994** \neq **435**

- Il numero 4.35 non ha una ***rappresentazione esatta*** nel sistema binario, proprio come $1/3$ non ha una rappresentazione esatta nel sistema decimale
- 4.35 viene rappresentato con un numero appena inferiore a 4.35, che, quando viene moltiplicato per 100, fornisce un numero appena un inferiore a 435

Intervalli numerici e precisione



- I numeri in **virgola mobile** hanno un **intervallo di variabilità molto ampio**
 - i **double**, ad esempio, hanno un valore assoluto massimo di circa **10^{308}** (**Double.MAX_VALUE**)
- ma soffrono di un altro importante problema, la **mancanza di precisione**, perché possono rappresentare “soltanto” 15 **cifre significative**
- Cosa significa “mancanza di precisione”?
 - significa che a volte le operazioni aritmetiche con numeri in virgola mobile **danno risultati inattesi...**

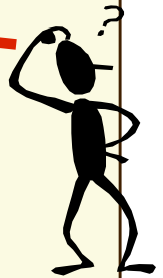


Intervalli numerici e precisione



```
public class DiscountTester
{
    public static void main(String[] args)
    {
        final double AMOUNT = 1.0e+17;
        final int DISCOUNT = 50;

        double doubleResult = AMOUNT - DISCOUNT;
        long longResult = ((long) AMOUNT) - DISCOUNT;
        System.out.println(doubleResult);
        // sbagliato per due unita` !!!!
        System.out.println(longResult);
        // questa volta e` giusto
    }
}
```



Assegnazioni con conversione

- In un'assegnazione, il **tipo** di dati dell'**espressione** e della **variabile** a cui la si assegna devono essere **compatibili**
 - se i tipi non sono compatibili, il compilatore segnala un **errore** (non sintattico ma **semantico**)
- I tipi **non** sono compatibili se provocano una **possibile perdita di informazione** durante la conversione
- L'assegnazione di un valore di tipo numerico intero a una variabile di tipo numerico in virgola mobile non può provocare perdita di informazione, quindi è ammessa

```
int intVar = 2;  
double doubleVar = intVar;
```



OK

Tipi di dati numerici incompatibili

```
double doubleVar = 2.3;
int intVar = doubleVar;
```

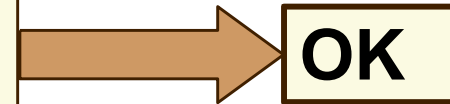
possible loss of
precision
found : double
required: int

- In questo caso si avrebbe una perdita di informazione, perché la (eventuale) **parte frazionaria** di un valore in virgola mobile non può essere memorizzata in una variabile di tipo intero
- Per questo motivo il compilatore non accetta un enunciato di questo tipo, segnalando l'errore semantico e interrompendo la compilazione

Conversioni forzate (cast)

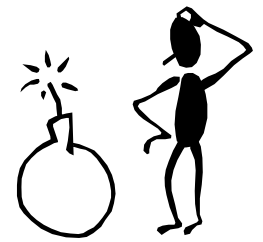
- Ci sono però casi in cui si vuole effettivamente ottenere la **conversione di un numero in virgola mobile in un numero intero**
- Lo si fa segnalando al compilatore l'intenzione **esplicita** di accettare l'eventuale perdita di informazione, mediante un **cast** ("forzatura")

```
double doubleVar = 2.3;  
int intVar = (int)doubleVar;
```



- Alla variabile **intVar** viene così assegnato il valore 2, la **parte intera** dell'espressione

Errori di arrotondamento



```
double f = 4.35F;  
int n = (int) (100 * f);  
System.out.println(n);
```

434 ≠ 435

- Come abbiamo visto, il numero 4.35 non ha una **rappresentazione esatta** nel sistema binario, proprio come 1/3 non ha una rappresentazione esatta nel sistema decimale
- Il cast fornisce quindi un risultato inatteso
 - 4.35 viene rappresentato con un numero appena un po' inferiore a 4.35, che, quando viene moltiplicato per 100, fornisce un numero appena un po' inferiore a 435, quanto basta però per essere troncato a 434
- È sempre meglio usare **Math.round** (che vediamo tra poco)

Conversioni con arrotondamento

- La conversione forzata di un valore in virgola mobile in un valore intero avviene con **troncamento**, *trascurando la parte frazionaria*
- Spesso si vuole invece effettuare tale conversione con **arrotondamento**, *convertendo all'intero più vicino*
- Ad esempio, possiamo **sommare 0.5 prima** di fare la conversione

```
double rate = 2.95;
int intRate = (int) (rate + 0.5);
System.out.println(intRate);
```



3

Conversioni con arrotondamento

- Questo semplice algoritmo per arrotondare i numeri in virgola mobile funziona però soltanto per numeri positivi, quindi non è molto valido...

```
double rate = -2.95;
int intRate = (int) (rate + 0.5);
System.out.println(intRate);
```



-2

- Un'ottima soluzione è messa a disposizione dal metodo round della classe Math della libreria standard, che funziona bene per tutti i numeri

```
double rate = -2.95;
int intRate = (int) Math.round(rate);
System.out.println(intRate);
```



-3

È tutto chiaro? ...

1. In quali situazioni il cast

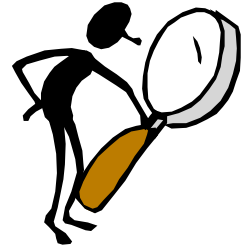
`(long) x`

produce un risultato diverso

dall'invocazione `Math.round(x)` ?

2. In che modo possiamo arrotondare al più vicino valore di tipo `int` il valore `x` di tipo `double`, sapendo che è minore di 2×10^9 ?

Altri tipi di dati numerici



- Come possiamo elaborare numeri interi o numeri in virgola mobile che non rientrino nel campo di variabilità di **long** o **double**? Come possiamo elaborare numeri in virgola mobile **con precisione arbitraria** (cioè con tutta la precisione necessaria per il problema in esame)?
- Il **pacchetto java.math** della libreria standard mette a disposizione due classi per rappresentare rispettivamente numeri interi (**BigInteger**) e numeri in virgola mobile (**BigDecimal**) che consentono di fare ciò, anche se in modo piuttosto lento e scomodo...



Altri tipi di dati numerici



```
import java.math.BigInteger;

public class BigNumbers
{   public static void main(String[] args)
    {   BigInteger a = new BigInteger("123456789");
        BigInteger b = new BigInteger("987654321");
        BigInteger c = a.multiply(b);
        System.out.println(c);
    }
}
```

121932631112635269

L'uso delle costanti

- Un programma per il cambio di valuta

```
public class Convert1
{
    public static void main(String[] args)
    {
        double dollars = 2.35;
        double euro = dollars * 0.72;
    }
}
```

- Chi legge il programma potrebbe legittimamente chiedersi quale sia il significato del “**numero magico**” **0.72** usato nel programma per convertire i dollari in euro...

L'uso delle costanti

- Così come si usano nomi simbolici descrittivi per le variabili, è opportuno assegnare **nomi simbolici** anche alle **costanti** utilizzate nei programmi

```
public class Convert2
{
    public static void main(String[] args)
    {
        final double EURO_PER_DOLLAR = 0.72;
        double dollars = 2.35;
        double euro = dollars * EURO_PER_DOLLAR;
    }
}
```

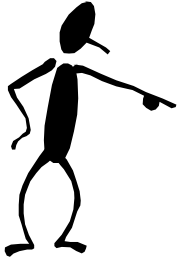
- Un primo **vantaggio** molto importante
 - **aumenta la leggibilità**

L'uso delle costanti

- Un ***altro vantaggio***: se il valore della costante deve cambiare (nel nostro caso, perché varia il tasso di cambio dollaro/euro), la modifica va fatta ***in un solo punto*** del codice!

```
public class Convert3
{
    public static void main(String[] args)
    {
        final double EURO_PER_DOLLAR = 0.72;
        double dollars1 = 2.35;
        double euro1 = dollars1 * EURO_PER_DOLLAR;
        double dollars2 = 3.45;
        double euro2 = dollars2 * EURO_PER_DOLLAR;
    }
}
```

Definizione di costante



- Sintassi:

```
final nomeTipo NOME_COSTANTE = espressione;
```

- Scopo: definire la costante NOME_COSTANTE di tipo nomeTipo, assegnandole il valore espressione, che non potrà più essere modificato
- Nota: il compilatore segnala come errore semantico il tentativo di assegnare un nuovo valore ad una costante, dopo la sua inizializzazione
- Di solito in Java si usa la seguente convenzione
 - i nomi di costanti sono formati da lettere maiuscole
 - i nomi composti si ottengono attaccando le parole successive alla prima con un carattere di sottolineatura

Operazioni aritmetiche

- L'operatore di **moltiplicazione** va sempre indicato **esplicitamente**, non può essere **sottinteso**
- Le operazioni di **moltiplicazione** e **divisione** hanno la **precedenza** sulle operazioni di **addizione** e **sottrazione**, cioè vengono eseguite prima
- È possibile usare **coppie di parentesi tonde** per indicare in quale ordine valutare sotto-espressioni

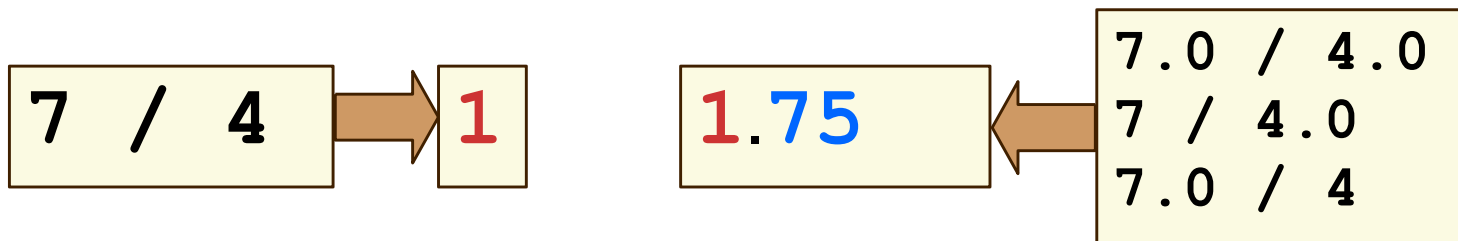
$$a + b / 2 \neq (a + b) / 2$$

- In Java non esiste il **simbolo di frazione**, le frazioni vanno espresse "**in linea**", usando l'operatore di divisione

$$\frac{a + b}{2} \rightarrow (a + b) / 2$$

Operazioni aritmetiche

- Quando **entrambi** gli operandi sono numeri **interi**, la **divisione** ha una caratteristica particolare, che può essere utile ma che va usata con attenzione
 - calcola il *quoziente intero*, scartando il *resto*!**

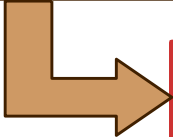


- Il resto della divisione tra numeri interi può essere calcolato usando l'operatore **%**, (questo simbolo non esiste in algebra, è stato scelto perché è simile all'operatore di divisione)

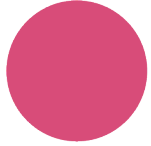


Divisione fra interi

```
public class Coins5
{   public static void main(String[] args)
    {   double euro = 2.35;
        final int CENT_PER_EURO = 100;
        int centEuro = (int) Math.round(euro *
CENT_PER_EURO);
        int intEuro = centEuro / CENT_PER_EURO;
        centEuro = centEuro % CENT_PER_EURO;
        System.out.print(intEuro);
        System.out.print(" euro e ");
        System.out.print(centEuro);
        System.out.println(" centesimi");
    }
}
```



2 euro e 35 centesimi



Funzioni più complesse: classe Math

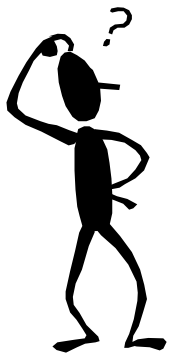
- Non esistono **operatori** per calcolare funzioni più complesse, come l'elevamento a potenza
- La classe **Math** della libreria standard mette a disposizione **metodi statici** per il calcolo di tutte le funzioni algebriche e trigonometriche, richiedendo parametri **double** e restituendo risultati **double**
 - **Math.pow(x, y)** restituisce x^y
 - (il nome **pow** deriva da **power**, potenza)
 - **Math.sqrt(x)** restituisce la radice quadrata di **x**
 - (il nome **sqrt** deriva da **square root**, radice quadrata)
 - **Math.log(x)** restituisce il logaritmo naturale di **x**
 - **Math.sin(x)** restituisce il seno di **x** espresso in radianti

Costanti della classe Math

- Nella classe **Math** sono definite alcune utili **costanti**

```
public final class Math
{
    ...
    public static final double PI =
        3.14159265358979323846;
    public static final double E =
        2.7182818284590452354;
}
```

- Sono **costanti** statiche, ovvero appartengono alla classe (approfondiremo in seguito)
- Tali costanti sono di norma **public** e per ottenere il loro valore si usa il nome della classe seguito dal punto e dal nome della costante, **Math.E**, oppure **Math.PI**



Combinare assegnazioni e aritmetica

- Abbiamo già visto come in Java sia possibile combinare in un unico enunciato *un'assegnazione* e *un'espressione aritmetica che coinvolge la variabile a cui si assegnerà il risultato*

```
totalEuro = totalEuro + dollars * 0.72;
```

- Questa operazione è talmente comune nella programmazione, che il linguaggio Java fornisce una **scorciatoia**

```
totalEuro += dollars * 0.72;
```

- che esiste per tutti gli operatori aritmetici

```
x = x * 2;
```

```
x *= 2;
```



Incremento di una variabile

- L'**incremento** di una variabile è l'operazione che consiste nell'**aumentarne il valore di uno**

```
int counter = 0;  
counter = counter + 1;
```

- Questa operazione è talmente comune nella programmazione, che il linguaggio Java fornisce un **operatore apposito per l'incremento**

```
counter++;
```

- e per il decremento

```
counter--;
```

È tutto chiaro? ...

1. Qual è il valore dell'espressione $1729/100$? E di $1729\%100$?
2. Perché questo enunciato non calcola la media tra $s1$, $s2$, ed $s3$?
`double average = s1 + s2 + s3 / 3;`
3. Come si esprime in notazione matematica la seguente espressione ?
`Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))`

Invocare metodi statici

```
double rate = -2.95;  
int intRate = (int)Math.round(rate);  
System.out.println(intRate);
```

- C'è una differenza sostanziale tra il metodo **round()** e il metodo **println()** già visto
 - **println()** agisce su un oggetto (**System.out**)
 - **round()** **non agisce** su un oggetto (**Math** è una classe)
- Il metodo **Math.round()** è un **metodo statico**
- La sintassi è identica, ma secondo la **convenzione**
 - i nomi di classi (**Math**, **System**) iniziano con una lettera maiuscola
 - i nomi di oggetti (**out**) e metodi (**println()**, **round()**) iniziano con una lettera minuscola
 - oggetti e metodi si distinguono perché solo i metodi sono sempre seguiti dalle parentesi tonde

Invocazione di un metodo statico

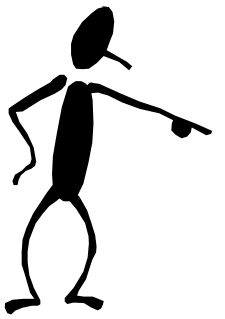
- Dalla documentazione della classe **java.lang.Math**:

```
public static long round(double a)
```

←
intestazione del
metodo round

- *public*: il metodo può essere invocato in qualsiasi classe
- *static*: il metodo è statico (altri metodi, ad esempio **println()**, o **translate()** della classe **Rectangle**, no)
 - un metodo statico si invoca usando il nome della classe in cui è definito, con la sintassi **NomeClasse.nomeMetodo**
 - es.: **Math.round(...)**
- *long*: tipo di dato restituito
 - è possibile che un metodo non restituisca dati, in questo caso il tipo del dato restituito è **void**
- *round*: nome o identificatore del metodo
- *double a*: parametro esplicito del metodo.

Invocazione di metodo statico



- Sintassi:

```
NomeClasse.nomeMetodo(parametri)
```

- Scopo: invocare il metodo statico ***nomeMetodo*** definito nella classe ***NomeClasse***, fornendo gli eventuali ***parametri*** richiesti
- Un metodo statico non viene invocato con un oggetto, ma con un nome di classe
 - Un metodo statico elabora o modifica solo i propri **parametri espliciti**



È tutto chiaro? ...

- Perché non si può invocare `x.pow(y)` per calcolare xy ?
- L'invocazione `System.out.println(4)` è l'invocazione di un metodo statico?

Il tipo di dati “stringa”

- I dati più importanti nella maggior parte dei programmi sono i **numeri** e le **stringhe**

- Una **stringa** è una **sequenza di caratteri**, che in Java (come in altri linguaggi) vanno **racchiusi tra virgolette**

"Hello"

- le virgolette **non** fanno parte della stringa
- Possiamo dichiarare, inizializzare, assegnare valori a **variabili di tipo stringa**

```
String name = "John";
```

```
name = "Michael";
```

- Diversamente dai numeri, le stringhe sono oggetti
 - Possiamo usare i **metodi** della classe String
 - ad esempio, il metodo **length()** restituisce la lunghezza di una stringa, cioè il numero di caratteri presenti in essa

```
String name = "John";  
int n = name.length();
```

4

Il tipo di dati “stringa”

- Il metodo **length** della classe **String** *non è un metodo statico*
- infatti *per invocarlo usiamo un oggetto della classe **String**, e non il nome della classe stessa*

```
// NON FUNZIONA!  
String s = "John";  
int n = String.length(s);
```

```
// FUNZIONA  
String s = "John";  
int n = s.length();
```

- Una stringa di **lunghezza zero**, che non contiene caratteri, si chiama **stringa vuota** e si indica con due caratteri virgolette **consecutivi**, senza spazi interposti

```
String empty = "";  
System.out.println(empty.length());
```

0

Estrazione di sottostringhe

Attenzione alla
minuscola!

- Per estrarre una sottostringa da una stringa si usa il metodo **substring**

```
String greeting = "Hello, World!";  
String sub = greeting.substring(0, 4);  
// sub contiene "Hell"
```

- il **primo** parametro di **substring** è la **posizione del primo carattere** che si vuole estrarre
- il **secondo** parametro è la **posizione successiva all'ultimo carattere** che si vuole estrarre

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

Estrazione di sottostringhe

- La **posizione** dei caratteri nelle stringhe viene **numerata a partire da 0** anziché da 1
 - in linguaggi precedenti, come il C e il C++, questa era un'esigenza **tecnica**, mentre in Java non lo è più e si è mantenuta questa caratteristica soltanto per **uniformità** con tali linguaggi molto diffusi
- Alcune cose da ricordare
 - la posizione dell'ultimo carattere corrisponde alla **lunghezza** della stringa **meno 1**
 - la differenza tra i due parametri di **substring** corrisponde alla **lunghezza** della sottostringa estratta



Estrazione di sottostringhe

- Il metodo **substring** può essere anche invocato con **un solo** parametro

```
String greeting = "Hello, World!";  
String sub = greeting.substring(7);  
// sub contiene "World!"
```

- In questo caso il parametro fornito indica la posizione del primo carattere che si vuole estrarre, e l'estrazione continua fino al termine della stringa

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

Estrazione di sottostringhe

- Cosa succede se si fornisce un **parametro errato** a **substring**?

```
// NON FUNZIONA!  
String greeting = "Hello, World!";  
String sub = greeting.substring(0, 14);
```

- Il programma viene compilato correttamente, ma viene generato un errore in esecuzione

```
Exception in thread "main"  
java.lang.StringIndexOutOfBoundsException  
String index out of range: 14  
at  
java.lang.String.substring(String.java:1444)  
at NomeClasse.main(NomeClasse.java:16)
```



Concatenazione di stringhe

- Per concatenare due stringhe si usa l'**operatore +**

```
String s1 = "eu";  
String s2 = "ro";  
String s3 = s1 + s2; // s3 contiene euro  
int euro = 15;  
String s = euro + s3; // s contiene "15euro"
```

- Il simbolo dell'operatore di concatenazione è identico a quello dell'operatore di addizione



- se una delle espressioni a sinistra o a destra dell'operatore **+** è una stringa, l'altra espressione viene **convertita automaticamente** in stringa e si effettua la concatenazione
- In alternativa per concatenare due stringhe è possibile utilizzare il metodo **concat** della classe String
 - Si può studiarne il funzionamento consultando la documentazione...

Concatenazione di stringhe

```
int euro = 15;  
String euroName = "euro";  
String s = euro + euroName;  
// s contiene "15euro"
```

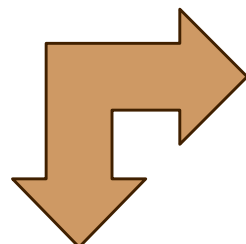
- Osserviamo che la concatenazione prodotta non è proprio quella che avremmo voluto, perché **manca uno spazio** tra **15** ed **euro**
 - l'operatore di concatenazione **non aggiunge spazi!**
- (**meno male**, diremo la maggior parte delle volte...)
- L'effetto voluto si ottiene così

```
String s = euro + " " + euroName;
```

Non è una stringa vuota, ma una stringa con **un solo carattere**, uno spazio (*blank*)

Concatenazione di stringhe

- La concatenazione è molto utile per ridurre il numero di enunciati usati per stampare i risultati dei programmi



```
int total = 10;  
System.out.print("Il totale è ");  
System.out.println(total);
```

```
int total = 10;  
System.out.println("Il totale è " + total);
```

- Bisogna fare attenzione a come viene gestito il concetto di “***andare a capo***” (cioè alla differenza tra **print** e **println**)

Altri metodi utili di String

- Un problema comune è la conversione di una stringa per ottenerne un'altra tutta in maiuscolo o in minuscolo
- La classe **String** mette a disposizione due metodi

- **toUpperCase** converte tutto in maiuscolo
- **toLowerCase** converte tutto in minuscolo

```
String s = "Hello";  
String ss = s.toUpperCase() + s.toLowerCase();  
// ss vale "HELLOhello"
```

- L'applicazione di questi metodi **non altera** il contenuto della stringa **s**, ma restituisce una **nuova stringa**
 - In generale, **nessun metodo** della classe **String** modifica l'oggetto con cui viene invocato!
 - si dice perciò che gli oggetti della classe **String** sono **oggetti immutabili**

Esempio

- Scriviamo un programma che genera la password per un utente, con la regola seguente
- si prendono le iniziali dell'utente, le si rendono minuscole e si concatena l'età dell'utente espressa numericamente

Utente: Sergio Canazza
Età: 18
⇒ Password: sc18

- (in realtà questa regola non è assolutamente da usare, perché è prevedibile e quindi poco sicura!)

Esempio

```
public class MakePassword
{
    public static void main(String[] args)
    {
        String firstName = "Sergio";
        String lastName = "Canazza";
        int age = 18; //slide da aggiornare :)

        // estrai le iniziali
        String initials = firstName.substring(0, 1)
            + lastName.substring(0, 1);
        // converti in minuscolo e concatena l'età
        String pw = initials.toLowerCase() + age;
        // stampa la password
        System.out.println("La password è " + pw);
    }
}
```

È tutto chiaro? ...

1. Se la variabile `s` di tipo `String` contiene il valore “Agent”, che effetto produce il seguente enunciato?

```
s = s + s.length() ;
```

2. Se la variabile `river` di tipo `String` contiene il valore “Mississippi”, che valori hanno le seguenti espressioni?

```
river.substring(1, 2)
```

```
river.substring(2, river.length() - 3)
```

Conversione di stringhe in numeri

- A volte si ha una stringa che contiene un valore numerico e si vuole assegnare tale valore a una variabile di tipo numerico, per poi elaborarlo

```
String password = "sc18";
String ageString = password.substring(2);
// ageString contiene "18"
// NON FUNZIONA!
int age = ageString;
```

incompatible types
found : java.lang.String
required: int

- Il compilatore segnala l'errore perché non si può convertire automaticamente una stringa in un numero, dato che ***non vi è certezza che il suo contenuto rappresenti un valore numerico***



Conversione di stringhe in numeri

- La conversione corretta si ottiene invocando il metodo statico **parseInt** della classe **Integer**

```
int age = Integer.parseInt(ageString);  
// age contiene il numero 18
```

- La conversione di un **numero in virgola mobile** si ottiene, analogamente, invocando il metodo statico **parseDouble** della classe **Double**

```
String numberString = "18.3";  
double number = Double.parseDouble(numberString);  
// number contiene il numero 18.3
```

- Integer e Double sono “**classi involucro**” dei tipi primitivi int e double

Conversione di stringhe in numeri

- ***Cosa succede se la stringa passata come argomento non contiene un numero?***
 - i metodi **Integer.parseInt** e **Double.parseDouble** lanciano un'**eccezione** di tipo **NumberFormatException** ed il programma termina segnalando l'errore
- Abbiamo già visto casi in cui il verificarsi di una eccezione arresta il programma
 - **StringIndexOutOfBoundsException** in **substring**
- Il meccanismo generale di segnalazione di errori in Java consiste nel "lanciare" (**throw**) un'**eccezione**
 - si parla anche di **sollevare** o **generare** un'eccezione
 - Vedremo più avanti il meccanismo di gestione delle eccezioni



Conversione di numeri in stringhe

- Per **convertire** un numero in stringa si può **concatenare il numero con la stringa vuota**

```
int ageNumber = 10;
String ageString = "" + ageNumber;
// ageString contiene "10"
```

- È però più elegante (e più comprensibile) utilizzare il metodo **toString** delle classi **Integer** e **Double**, rispettivamente per numeri interi e numeri in virgola mobile

```
int ageNumber = 10;
String ageString = Integer.toString(ageNumber);
```

Caratteri in una stringa

- Sappiamo già come estrarre sottostringhe da una stringa, con il metodo **substring**
- A volte è necessario estrarre ed elaborare **sottostringhe** di dimensioni minime, cioè **di lunghezza unitaria**
 - una stringa di lunghezza unitaria contiene **un solo carattere**, che **può essere memorizzato in una variabile di tipo char** anziché in una stringa
 - il tipo **char** in Java è **un tipo di dato fondamentale** come i tipi di dati numerici e il tipo **boolean**, cioè **non è una classe**

Caratteri in una stringa

- La presenza del tipo di dati **char** non è strettamente necessaria in Java (ed è anche per questo motivo che non l'avevamo ancora studiato)
- infatti, **ogni** elaborazione che può essere fatta su variabili di tipo **char** potrebbe essere fatta su stringhe di lunghezza unitaria
- L'uso del tipo **char** per memorizzare stringhe di lunghezza unitaria è però importante, perché
 - una variabile di tipo **char** occupa **meno spazio** in memoria di una stringa di lunghezza unitaria
 - le **elaborazioni** su variabili di tipo **char** sono **più veloci**

Caratteri in una stringa

- Il metodo **charAt** della classe **String** restituisce il singolo carattere che si trova nella posizione indicata dal parametro ricevuto

```
String s = "John";  
char ch = s.charAt(2); // ch contiene 'h'
```

- la convenzione sulla numerazione delle posizioni in una stringa è la stessa usata dal metodo **substring**

```
String s = "John";  
for (int i = 0; i < s.length(); i++)  
{  
    char ch = s.charAt(i);  
    // elabora ch  
}
```



- Una struttura di controllo che si usa spesso è l'elaborazione di tutti i caratteri di una stringa

Elaborazioni su variabili char

- Come si può **elaborare** una variabile di tipo **char**?
- Una variabile di tipo **char** può anche essere confrontata con una **costante di tipo carattere**

```
char ch = 'x';
```

- una costante di tipo carattere è **un singolo carattere** racchiuso tra **singoli** apici (apostrofo)

```
char ch = '\u00E9'; // carattere 'è'  
char nl = '\n'; // carattere di "andata a capo"
```

- Il carattere può anche essere una “**sequenza di escape**”

```
char ch = 'x';  
System.out.println(ch); // stampa 'x' e va a capo
```

- Una variabile carattere può essere **stampata** passandola a **System.out.print** o **System.out.println**

Caratteri e l'operatore +

- Un char può essere **concatenato a una stringa** con l'operatore di **concatenazione** + (viene convertito in stringa, con le stesse regole della conversione dei tipi numerici)
- Se invece i due operandi di + sono entrambi caratteri
 - Vengono automaticamente convertiti (“**promossi**”) in **int**
 - L'operatore + indica una normale **somma tra interi**

```
char a = 'a';  
char b = 'b';  
int intc = a + b;  
char c = (char) (a + b);  
System.out.println("intc: " + intc);  
System.out.println("c: " + c);
```

intc: 195
c: Ã



Visualizzazione di caratteri non-ASCII

- Java gestisce correttamente i caratteri Unicode, alcuni sistemi operativi **no**.
- Se un programma Java stampa una stringa che contiene un carattere che non fa parte del codice ASCII, al suo posto possono venire stampati caratteri strani
- Verificate il vostro sistema con questo programma

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

- Per evitare il problema, si consiglia di
 - **non usare** lettere accentate nei messaggi visualizzati dai programmi (usare, in alternativa, l'apostrofo).
 - Usare **solo** caratteri ASCII

Sequenze di “escape”

- Proviamo a stampare una stringa che **contiene** delle virgolette

```
Hello, "World"!
```

```
// NON FUNZIONA!
```

```
System.out.println("Hello, "World"!");
```

- Il compilatore identifica le seconde virgolette come la fine della prima stringa **"Hello, "**, ma poi non capisce il significato della parola **World**
- Basta inserire una barra rovesciata **** (*backslash*) **prima** delle virgolette **all'interno** della stringa

```
System.out.println("Hello, \"World\"!");
```

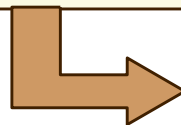
Sequenze di “escape”

```
// FUNZIONA!
```

```
System.out.println("Hello, \"World\"!");
```

- Il carattere *backslash* all'interno di una stringa non rappresenta se stesso, ma si usa per codificare altri caratteri che sarebbe **difficile** inserire in una stringa, per vari motivi (**sequenza di escape**)
- Allora, come si fa ad inserire veramente un carattere *backslash* in una stringa?
- si usa la sequenza di escape ****

```
System.out.println("File C:\\autoexec.bat");
```

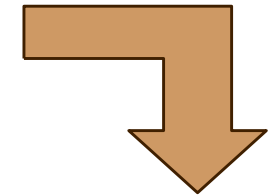


```
File C:\autoexec.bat
```

Sequenze di “escape”

- Le sequenze di escape si usano per inserire caratteri speciali o simboli che non si trovano sulla tastiera

```
System.out.println("Perch\u00E9?");
```



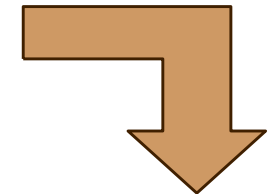
Perché?

- La sequenza di escape **\U00E9** indica
- la codifica Unicode del carattere
- Un'altra frequente sequenza di escape è **\n**, che rappresenta il carattere di “nuova riga” o “a capo”

```
System.out.println("*\n**\n***");
```

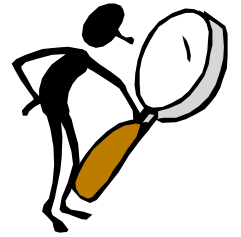


```
System.out.println("*");  
System.out.println("**");  
System.out.println("***");
```



*
**

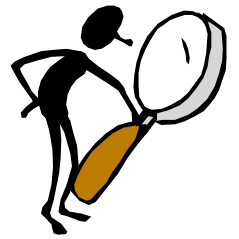
Escape e caratteri di controllo



- Oltre a `\n`, anche altre sequenze di escape possono essere usate per rappresentare **caratteri di controllo**
- Ovvero caratteri **Unicode** che **non rappresentano simboli scritti**
- Ma che fanno parte integrante di un flusso di testo, come tutti gli altri caratteri

SEQ.ESCAPE	NOME	COD.UNICODE
<code>\n</code>	<code>newline</code>	<code>\u000A</code>
<code>\t</code>	<code>tab</code>	<code>\u0009</code>
<code>\b</code>	<code>backspace</code>	<code>\u0008</code>
<code>\r</code>	<code>return</code>	<code>\u000D</code>
<code>\f</code>	<code>formfeed</code>	<code>\u000C</code>

Caratteri di controllo

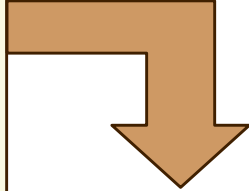


- I primi 32 caratteri nella codifica **Unicode** sono tutti caratteri di controllo
 - Il cosiddetto **insieme C0 di ASCII (e Unicode)**
 - Per chi e' interessato
 - http://en.wikipedia.org/wiki/C0_and_C1_control_codes
 - http://en.wikipedia.org/wiki/Control_character
- Vedremo altri importanti caratteri di controllo
 - **ETX** (End-of-TeXt), immesso da tastiera con **<CTRL>+C**
 - usato per interrompere l'esecuzione di un programma
 - **EOT** (End-Of-Transmission, **<CTRL>+D**) e **SUB** (SUBstitute, **<CTRL>+Z**)
 - Usati per segnalare: ^D la fine dell'input (per esempio di un file); ^Z mette in pausa (manda in background)

Formattazione di numeri

- Non sempre il **formato standard** per stampare numeri corrisponde ai nostri desideri

```
double total = 3.50;  
final double TAX_RATE = 8.5; // aliquota  
d'imposta in percentuale  
double tax = total * TAX_RATE / 100;  
System.out.println("Total: " + total);  
System.out.println("Tax: " + tax);
```



Total: 3.5
Tax: 0.2975

- Ci piacerebbe di più visualizzare i numeri
 - Con due cifre decimali
 - Incolonnati

Total: 3.50
Tax: 0.30

Formattazione di numeri

- Java fornisce il metodo **printf**

- Il primo parametro esplicito di **printf** è una **stringa di formato** che contiene dei caratteri da stampare e degli **specificatori di formato**

- Ogni specificatore di formato comincia con il carattere %

- I parametri successivi sono i **valori da visualizzare** secondo i formati specificati

```
System.out.printf("Total:%5.2f", total)
```

- Produce:

Total: 3.50

Spazio

- **%5.2f** è lo specificatore di formato: numero **in virgola mobile** (%f) formato da **5 caratteri** (compreso il punto!) con **due cifre dopo la virgola**
- Questo formato viene applicato alla variabile total, che è il secondo parametro del metodo

Formattazione di numeri

Esempio printf:

```
double a = 1, b = 2;
```

```
double s = a + b;
```

```
//Voglio stampare la somma —> “1 + 2 = 3”
```

```
PRINTLN —> System.out.println(a + “ + ” + b + “ = ” s);
```

```
PRINTF —>
```

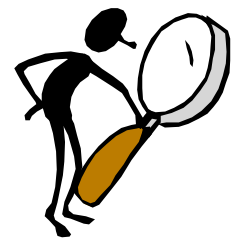
```
System.out.printf(“%f + %f = %f %n”, a, b, s);
```

```
System.out.printf(“%5.2f + %5.2f = %5.2f %n”, a, b, s);
```

```
System.out.printf(Locale.US, “%5.2f + %5.2f = %5.2f %n”, a, b,  
s);
```

```
1.0+2.0=3.0
1,000000 + 2,000000 = 3,000000
1,00 + 2,00 = 3,00
1.00 + 2.00 = 3.00
```


Formattazione di numeri



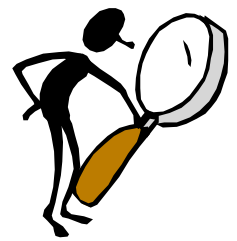
- **Tipi** di formato e **modifiers** di formato:

Codice	Tipo	Esempio
d	Intero decimale	123
x	Intero esadecimale	7B
o	Intero ottale	173
f	Virgola mobile	12.30
e	Virgola mobile esponenziale	1.23e+1
g	Virgola mobile generico (notazione esponenziale per i numeri molto grandi o molto piccoli)	12.3
s	Stringa	Tax:
n	Fine riga indipendente dalla piattaforma	

Codice	Significato	Esempio
-	Allinea a sinistra	1.23 seguito da spazi
0	Mostra gli zeri iniziali	001.23
+	Mostra il segno più per numeri positivi	+1.23
(Racchiude tra parentesi i numeri negativi	(1.23)
,	Mostra il separatore di migliaia	12,300
^	Usa lettere maiuscole	12.3E+1



Caratteri di fine riga



- Diversi sistemi operativi “capiscono” diversi caratteri di fine riga
 - Sistemi Unix usano il carattere newline (o line-feed): `\n`
 - DOS usa la sequenza carriage return-newline: `\r\n`
- Per essere sicuri che la fine della riga sia riconosciuta da qualsiasi sistema operativo possiamo usare il metodo **printf** con il formato `%n`

```
System.out.printf("%n Total:%5.2f", total)
```

- produce

```
Total: 3.50
```

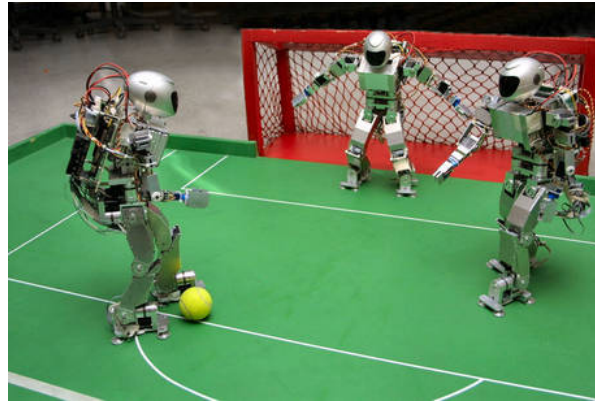
Nuova riga

I dati in ingresso ai programmi

- I programmi visti finora non sono molto utili, visto che eseguono ***sempre la stessa elaborazione a ogni esecuzione***
- Il programma **Coins1** rappresenta sempre il medesimo borsellino...
 - se si vuole che calcoli il valore contenuto in un diverso borsellino, è necessario modificare il codice sorgente (in particolare, le inizializzazioni delle variabili) e compilarlo di nuovo
- I programmi utili hanno bisogno di ***ricevere dati in ingresso*** dall'utente

L'input standard dei programmi

- Il modo più semplice e immediato per fornire dati in ingresso ad un programma consiste nell'**utilizzo della tastiera**
 - altri metodi fanno uso del mouse, del microfono, sensori...



- Abbiamo visto che tutti i programmi Java hanno accesso al proprio **output standard**, tramite l'oggetto **System.out** di tipo **PrintStream**
- Analogamente, l'interprete Java mette a disposizione dei programmi in esecuzione il proprio input standard (**flusso di input**), tramite l'oggetto **System.in** di tipo **InputStream**

La classe **Scanner**

- Sfortunatamente, la classe **InputStream** *non possiede metodi comodi* per la ricezione di dati numerici e stringhe
 - **PrintStream** ha invece il *comodissimo* metodo **print**
- Per ovviare a questo inconveniente, Java 5.0 ha introdotto la classe **Scanner**
 - Un oggetto di tipo **Scanner** consente di leggere da qualsiasi flusso di ingresso (ad es. un file)
 - Noi cominciamo a usarlo per leggere dati in ingresso da tastiera ricevuti tramite l'oggetto **System.in**

Usare la classe Scanner

Sono due oggetti diversi!!

- Per leggere dallo standard input bisogna creare un oggetto di tipo **Scanner**, usando la sintassi consueta

```
Scanner in = new Scanner(System.in);
```

- Il parametro esplicito del costruttore di **Scanner** è **System.in**
- L'oggetto in di tipo **Scanner** è “**agganciato**” allo standard input
- Dato che la classe **Scanner** non fa parte del pacchetto **java.lang**, ma del pacchetto **java.util**, è necessario **importare esplicitamente** la classe all'interno del file java che ne fa uso

```
import java.util.Scanner;
```

- Quando non si usa più l'oggetto di classe Scanner e' bene chiuderlo:

```
in.close();
```

I metodi nextInt e nextDouble

- Come si fa ad **acquisire valori numerici** da standard input?
- Numero intero: metodo **nextInt()**

```
int number = in.nextInt();
```

- Numero in virgola mobile: metodo **nextDouble()**

```
double number = in.nextDouble();
```

- Durante l'esecuzione del metodo (**nextInt** o **nextDouble**) *il programma si ferma ed attende* l'introduzione dell'input da tastiera, che termina quando l'utente batte il tasto **Invio**
- **nextInt** restituisce un valore numerico di tipo **int**
- **NextDouble** restituisce un valore numerico di tipo **double**
 - cosa succede se l'utente non digita un numero intero (o un numero double) sulla tastiera ??
Provare!!

Esempio

```
import java.util.Scanner;

public class Coins6
{
    public static void main(String[] args)
    {
        Scanner ingresso = new Scanner(System.in);
        System.out.println("Quante lire?");
        int lit = ingresso.nextInt();
        System.out.println("Quanti euro?");
        double euro = ingresso.nextDouble();
        System.out.print("Valore totale in euro ");
        System.out.printf("%5.2f%n", euro + lit/1936.27);
        System.out.println(euro + lit/1936.27);
        ingresso.close();
    }
}
```

Quante lire?
25000
Quanti euro?
34,5
Valore totale in euro 47,41
47.41142247723716

I metodi next e nextLine

- Come si fa ad **acquisire stringhe** da standard input?

- Parola

- ovvero una stringa delimitata dai **caratteri di spaziatura** space (**SP**), tab (**\t**), newline (**\n**), carriage-return (**\r**)

- metodo **String next()**

```
String state = in.next();
```

- Riga

- (ovvero una stringa delimitata dai caratteri **\n** o **\r**):

- metodo **String nextLine()**:

```
String city = in.nextLine();
```

Esempio

- Inseriamo tre dati di input su **tre righe** diverse

```
import java.util.Scanner;

public class MakePassword2
{   public static void main(String[] args)
    {   Scanner in = new Scanner(System.in);
        System.out.println("Inserire il nome");
        String firstName = in.nextLine();
        System.out.println("Inserire il cognome");
        String lastName = in.nextLine();
        System.out.println("Inserire l'eta' ");
        int age = Integer.parseInt(in.nextLine());
        in.close();
        String initials = firstName.substring(0, 1)
            + lastName.substring(0, 1);
        String pw = initials.toLowerCase() + age;
        System.out.println("La password e' " + pw);
    }
}
```

Esempio

- Inseriamo tre dati di input sulla **stessa riga**

```
import java.util.Scanner;

public class MakePassword3
{   public static void main(String[] args)
    {   Scanner in = new Scanner(System.in);
        System.out.println("Inserire nome, cognome, eta`" +
                            "sulla stessa riga");

        String firstName = in.next();
        String lastName = in.next();
        int age = in.nextInt();
        in.close();
        String initials = firstName.substring(0, 1)
            + lastName.substring(0, 1);
        String pw = initials.toLowerCase() + age;
        System.out.println("La password e' " + pw);
    }
}
```

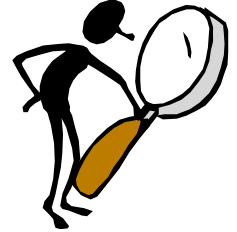
Esempio

- E qui cosa succede? I tre input vengono rilevati **sia** quando vengono inseriti sulla stessa riga, **sia** quando vengono inseriti su tre righe diverse

```
import java.util.Scanner;

public class MakePassword4
{   public static void main(String[] args)
    {   Scanner in = new Scanner(System.in);
        System.out.println("Inserire il nome");
        String firstName = in.next();
        System.out.println("Inserire il cognome");
        String lastName = in.next();
        System.out.println("Inserire l'eta'");
        int age = in.nextInt();
        in.close();
        String initials = firstName.substring(0, 1)
            + lastName.substring(0, 1);
        String pw = initials.toLowerCase() + age;
        System.out.println("La password e' " + pw);
    }
}
```

Scanner e localizzazione



- La classe **Scanner** prevede la possibilità di riconoscere numeri formattati secondo diverse “usanze locali”
- Nel corso noi useremo la convenzione **anglosassone** che prevede l’uso del carattere di separazione ‘.’ tra parte intera e parte frazionaria nei numeri in virgola mobile
- Per definire un oggetto **Scanner** che rispetta questa convenzione dovremo **importare anche la classe Locale** e scrivere

```
import java.util.Scanner;  
import java.util.Locale;  
...  
Scanner in = new Scanner(System.in) ;  
in.useLocale(Locale.US) ;  
...
```



È tutto chiaro? ...

- Perché non si possono leggere dati in ingresso direttamente tramite `System.in` ?
- Se `in` è un oggetto di tipo `Scanner` che legge da `System.in` e il nostro programma invoca

```
String name = in.next();
```

Qual è il valore di `name` se l'utente digita la seguente stringa?

```
John Q. Public
```
