

Corso: Paradigmi di Programmazione

Paola Giannini

Implementazione/Specifiche dei linguaggi di programmazione

Implementazione dei linguaggi di programmazione

I programmi possono essere eseguiti in due modalità:

- **Interpretati**

- un altro programma o componente hardware, detto **interprete**, esamina una per una le istruzioni del programma eseguito, e opera in modo da realizzarne gli effetti

- **Compilati**

- un programma detto **compilatore** traduce il programma in un programma analogo scritto in un altro linguaggio, che poi viene a sua volta compilato o interpretato

Caratteristiche dei linguaggi compilati/interpretati

- I linguaggi interpretati

- la dichiarazione di una variabile **non specifica il tipo**; il controllo di tipo avviene a tempo di esecuzione
- il **ciclo di programmazione è più breve** (non richiedendo la compilazione)
- l'**esecuzione è più lenta**
- il codice è **portabile su architetture diverse**, basta avere un interprete

- I linguaggi compilati

- la dichiarazione di una variabile **specificava il suo tipo** il cui uso corretto è testato a tempo di compilazione
- il **ciclo di programmazione è più lungo** e richiede un **programmatore più esperto**
- l'**esecuzione è più efficiente**
- il codice ha come target un'architettura specifica

Python è interpretato, mentre Java usa un modello misto

- i programmi Java sono compilati in un linguaggio intermedio detto **bytecode**
 - il bytecode è quindi interpretato da un interprete detto **Java Virtual Machine (JVM)**
 - alternativamente, il bytecode può essere compilato in linguaggio macchina che infine viene interpretato dalla CPU

Java usa un modello misto,

- questo permette di avere la **robustezza** di un linguaggio compilato e la **portabilità** di uno interpretato
- il bytecode è sufficientemente ad alto livello (è anch'esso tipato) per permettere controlli che rendono l'esecuzione dei programma Java sicura.

Come si definiscono i linguaggi

- I linguaggi di programmazione devono essere estremamente precisi
- In particolare devono essere **precisamente definite**:
 - **la sintassi** del linguaggio, cioè quali sequenze di caratteri (parole chiave, simboli, ecc.) costituiscono programmi validi
 - **la semantica** del linguaggio, cioè qual'è l'effetto dell'esecuzione di ogni programma valido
- Noi indicheremo formalmente la sintassi, ma useremo spiegazioni informali per la semantica
 - Java ha comunque una semantica formale, mentre quella di Python non è definita formalmente!

Grammatiche EBNF (1)

- EBNF: Extended Backus-Naur Form
- Una **grammatica** è definita da un insieme di produzioni
- Ogni **produzione** ha la forma

Classe ::= Espansione

- *Classe* è un **simbolo non-terminale**
- *Espansione* è una sequenza di **simboli terminali** e non-terminali
- Nell'*Espansione* simbolo | indica alternativa.

Grammatiche EBNF (2)

Alcuni simboli, che useremo come **apici** hanno un significato speciale:

- ? indica un elemento opzionale (0 o 1 occorrenza)
- + indica un elemento ripetuto (1 o più occorrenze)
- * indica un elemento opzionale ripetuto (0 o più occorrenze)
- le parentesi tonde (e) servono per raggruppare. Useremo (e) (different font) per indicare le parentesi tonde come simboli terminali della grammatica

Esempio di specifica EBNF

Vogliamo definire la sintassi di un indirizzo postale.

Indirizzo ::= *Destinatario Luogo Citta*
Destinatario ::= *Nome*⁺
Luogo ::= (*via* | *piazza* | *largo*) *Nome*⁺ *Numero*
Citta ::= *CAP Numero*, *Nome*
Nome ::= (*a* | ... | *z* | *A* | ... | *Z*)⁺
Numero ::= (*1* | ... | *9*)(*0* | ... | *9*)*

- Le **classi sintattiche**: *Indirizzo*, *Destinatario*, *Luogo*, *Citta*, *Nome*, *Numero* denotano l'insieme di stringhe che hanno la forma descritta dalle loro produzioni
- I simboli terminali: *via*, *piazza*, *largo*, *CAP*, *a*, ..., *z*, *A*, ..., *Z*, *0*, ..., *9* e , (la virgola) denotano **una stringa**.

Classi sintattiche dei linguaggi di programmazione

Nella definizione dei linguaggi incontreremo alcune classi particolarmente significative

- **Espressioni**: denotano un valore, tipicamente ottenuto tramite calcoli
- **Dichiarazioni**: definiscono e allocano spazio in memoria per contenere valori (variabili)
- **Comandi o Istruzioni**: indicano quali operazioni devono essere eseguite e in che ordine

Useremo i puntini ... nelle produzioni per indicare che la produzione non è completa

Corso: Paradigmi di Programmazione

Paola Giannini

Introduzione a Java

I linguaggi ad oggetti precursori di Java

- 1967 : **Simula 67** (nasce dall'Algol), un linguaggio ad oggetti per la simulazione discreta di sistemi. Il paradigma ad oggetti è particolarmente adatto a rappresentare il dominio delle simulazioni. Il linguaggio supporta co-routine. Il linguaggio è **compilato**. Le classi sono tipi e si possono estendere.
- 1980 : nasce **Smalltalk**, linguaggio in cui tutto è un oggetto (gli interi, i booleani, i blocchi di codice) e la computazione si svolge mandando messaggi ad oggetti i quali eseguono in risposta un metodo. Il linguaggio è **interpretato** (non ci sono tipi). Smalltalk ha un ambiente di programmazione (oggi si direbbe IDE) e non necessita una sintassi esplicita.
- 1991 : nasce **“green”** alla Sun Microsystems (il predecessore di Java). L'idea è di usarlo negli elettrodomestici. **Combinazione di compilazione e interpretazione.**
- 1994 : l'ideatore di “green” James Gosling, si rende conto che il nuovo linguaggio poteva avere grosse potenzialità per applicazioni client/server.

...e Java

1995 : nasce ufficialmente Java ed un browser HotJava capace di scaricare programmi dalla rete e eseguirli (Applet).

1996 : v1.0, poi classi innestate.

1998 : v1.2, collections, e swing (GUI indipendenti dalla piattaforma) pattern MVC (introdotto da Smalltalk)

2002 : v1.4, asserzioni, XML

2005 : v1.5, classi generiche, estensione ciclo for, autoincapsulamento, enumerazioni.....

..... : .inferenza di tipo (2011), lambda-astrazioni (2014).....

Requisiti software per l'esecuzione di Java

- Per sviluppare programmi Java abbiamo bisogno almeno di un editore di testi (per scrivere i programmi), di un compilatore e di un interprete.
- Il Java Standard Edition Development Kit (JDK), scaricabile dal sito

<https://www.oracle.com/technetwork/java/index.html>

contiene compilatore e interprete standard, cui faremo riferimento

- Si può usare (come faremo noi) un ambiente integrato di sviluppo (IDE) come eclipse

- Per eseguire i programmi che vedremo dobbiamo copiare nella directory che li contiene la directory jbook, disponibile sul sito del corso (fornisce metodi per fare I/O in modo semplice).
- Un semplice programma Java è costituito da una classe contenente un metodo chiamato main
- Una classe deve essere scritta in un file avente lo stesso nome della classe, con suffisso .java

Il programma Hello

- Deve essere contenuto nel file Hello.java
- Lo analizzeremo in seguito...

```
1. /*
2.      Un programma che chiede il tuo nome e ti saluta
3. */
4.
5. import jbook.util.Input;
6.
7. public class Hello {
8.     public static void main(String[] args){
9.         System.out.print("Come ti chiami? "); //stampa
10.        String persona;
11.        persona = Input.readString(); //legge
12.        System.out.println("Ciao " + persona + "!\n"); //stampa
13.    }
14.}
```

Come eseguire il programma Hello

- Per compilare il programma Hello eseguiamo `javac Hello.java` in un interprete di comandi (shell Linux, o interprete DOS sotto Windows)
- Se tutto è corretto viene generato il file `Hello.class` contenente il codice oggetto (bytecode)
- Per eseguirlo, lanciamo il comando `java Hello`
- In un IDE come eclipse possiamo usare gli appositi pulsanti invece di javac e java

Esempio di esecuzione

- La Java Virtual Machine invocata dal comando `java` esegue in sequenza i comandi del metodo `main`

```
8.      System.out.print("Come ti chiami? "); //stampa
9.      String persona;
10.     persona = Input.readString();        //legge
11.     System.out.println("Ciao " + persona + '!'); //stampa
```

Una possibile interazione:

```
> Come ti chiami? Maria
> Ciao Maria!
```

La compilazione

- La compilazione è divisa in varie fasi:
 - analisi lessicale
 - analisi sintattica
 - generazione del codice (bytecode)
- L'analisi lessicale e sintattica controllano
 - che il programma sia sintatticamente corretto, cioè che sia **generabile dalla grammatica di Java**
 - che rispetti alcune regole di semantica statica, come la **consistenza dei tipi e la visibilità dei nomi**

- L'analisi lessicale trasforma il programma in una sequenza di **token**, entità elementari del linguaggio. I commenti vengono ignorati.
- I token, divisi in 5 categorie, sono generati dalle produzioni della grammatica lessicale di Java

*Token ::= ParolaChiave | Separatore | Operatore
| CostanteLetterale | Identificatore*

- Gli identificatori sono opportune sequenze di lettere e numeri, usate come nomi di entità dei programmi. Java come il C è case sensitive!

Parole chiave

- Sono nomi riservati con significato predefinito: verranno introdotti progressivamente

ParolaChiave ::=

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

Operatori e Separatori

- Ci sono operatori aritmetici, logici, di confronto, di assegnamento e altri: li vedremo man mano

Operatore ::=

=	>	<	!	~	?	:	==	<=				
>=	!=	&&		++	--	+	-	*				
/	&		^	%	<<	>>	>>>	+=				
-=	*=	/=	&=	=	^=	%=	<<=	>>=				
>>>=												

- I separatori comprendono segni di interruzione e parentesi

Separatore ::= (|) | [|] | { | } | ; | , | .

Le costanti letterali

- Sono sequenze di caratteri che rappresentano valori di un certo tipo di dati

*CostanteLetterale ::= LettIntero | LettVirgolaMobile | LettCarattere
| LettStringa | true | false | null*

- Per esempio, sono costanti letterali corrette

- gli interi: 345 e -12345678
- i numeri in virgola mobile: 34.56 e 3456e3
- i caratteri: 'c' e '\n'
- la stringa: "Come ti chiami?"

L'analisi sintattica

L'analisi sintattica

- controlla che la sequenza di token prodotta dall'analisi lessicale sia corretta, cioè generabile dalla grammatica di Java
- riconosce i costrutti del linguaggio usati nel programma
- controlla che alcune regole di semantica statica siano rispettate, ad esempio, una variabile viene usata dopo la sua dichiarazione
- produce un albero di derivazione che verrà usato per generare il codice oggetto

Analisi sintattica del programma Hello

- Vediamo alcuni costrutti riconosciuti dall'analisi

```
5 import jbook.util.Input;
6
7 public class Hello {
8     public static void main(String[] args){
9         System.out.print("Come ti chiami? ");
10        String persona;
11        persona = Input.readString();
12        System.out.println("Ciao " + persona + "!");
13    }
14 }
```

Direttiva di importazione

Dichiarazione della classe Hello

Dichiarazione del metodo main

Dichiarazione di variabile

Assegnamento

Comandi di stampa (invocazioni di metodo)

L'esecuzione del programma Hello

Lanciando il comando `java Hello` l'interprete Java esegue in sequenza le istruzioni del metodo `main`

- `System.out.print("Come ti chiami? ");`

Stampa su terminale `Come ti chiami?`, usando un metodo di libreria

- `String persona;`
`persona = Input.readString();`

Dichiara la variabile `persona`, quindi legge da tastiera una sequenza di caratteri e la assegna a `persona`

- `System.out.println("Ciao " + persona + '!');`

Stampa `Ciao`, seguito dalla stringa letta, seguito da `!`

Corso: Paradigmi di Programmazione

Paola Giannini

Variabili e organizzazione dei nomi

Le Variabili

In Java le variabili devono essere dichiarate. La dichiarazione di una variabile specifica

- un nome
- un tipo
- alcuni attributi (definiti tramite i modificatori)
- un contenuto
- Il tipo determina
 - l'insieme dei valori ammissibili (assegnabili)
 - l'occupazione di memoria

Dichiarazione di variabili

- Per il momento consideriamo solo **variabili locali**
 - dichiarate nel corpo di un metodo
 - un solo modificatore: **final**
 - (più avanti vedremo variabili statiche e variabili d'istanza)
- Riserva lo spazio in memoria necessario
- Fissa nome, tipo e attributi (definiti tramite i modificatori)
- Determina **l'ambito di visibilità**, lo **scope**
 - dalla dichiarazione fino alla fine del blocco (un blocco è delimitato da parentesi graffe)

Dichiarazione di variabili: sintassi

- Una porzione (significativa) della sintassi della dichiarazione di variabili

CmdDicVarLoc ::= *DicVarLoc* ;
DicVarLoc ::= *ModVarLoc*? *Tipo* *DichVars*
ModVarLoc ::= **final** | ...
DichVars ::= *DichVar* (, *DichVar*)*
DichVar ::= *Id* (= *EsprIniVar*)?
EsprIniVar ::= *Espressione* | *EsprIniArray*

Dichiarazione di variabili: esempi

- **Semplice** (una sola variabile)
- **Multipla** (più variabili, tutte dello stesso tipo)
- **Con inizializzazione** (dichiara e assegna valore)
- Esempi

```
String persona ; // semplice
int i, j, k ; // multipla
int lato = 5,
    area = lato * lato ; // multipla con iniz.
```

Nomi di variabili

- Identifieri ammissibili:
 - sequenze alfanumeriche o contenenti anche _ (underscore)
 - che non siano parole riservate del linguaggio
 - che non comincino con una cifra
- Spazi, punteggiatura, parentesi, apici e asterischi: non consentiti!
- Le lettere accentate: consentite, ma sconsigliate
- Avvertenze
 - Per variabili particolarmente rilevanti dichiarazione semplice preferibile a quella multipla
 - Il compilatore non consente di dichiarare due variabili con lo stesso nome se il loro ambito si sovrappone (es: nella dichiarazione multipla tutti i nomi dichiarati devono essere diversi)

Convenzioni

- I nomi composti da una sola parola sono minuscoli
 - es: nome, indirizzo
- Se composti da più parole: la prima lettera di ogni parola successiva alla prima è maiuscola
 - es: telefonoAbitazione, telefonoUfficio
- Il carattere _ (underscore) è usato solo nei nomi di costanti composti da più parole
- Sul sito avete un link alle
 - convenzioni usate dagli sviluppatori di Google per la scrittura del codice in Java e un link ad
 - un video gentilmente condiviso dalla prof.ssa Lavinia Egidi in cui si illustrano le convenzioni più rilevanti per noi.

Tipi

- Tipi di dati **primitivi**
 - detti anche **tipi elementari** o **tipi base** (rendono Java un linguaggio **OO impuro**)
- Tipi di dati **riferimento**
 - detti anche **tipi oggetto**

Tipo ::= *TipoPrim* | *TipoRif*

TipoPrim ::= **byte** | **short** | **int** | **long** | **float** | **double** | **char** | **boolean**

TipoRif ::= *IdQual* | ...

IdQual ::= (*Id*.)* *Id*

Tipi riferimento

- Per il momento consideriamo solo **nomi qualificati**
- Per esempio, le classi
 - `String` per stringhe (sequenze di caratteri)
 - `java.awt.Point` (per punti con coordinate intere)

Il contenuto di una variabile

- Il contenuto di una variabile (in un dato istante) è il **valore presente nella memoria allocata per essa**
 - le variabili di tipo riferimento sono allocate nell'**heap**
- Java è **type-safe**, nel senso di garantire che:
 - per variabili di tipo primitivo il contenuto sia un valore ammissibile di quel tipo
 - per variabili di tipo riferimento il contenuto sia un riferimento a un oggetto (di tipo compatibile), oppure la costante speciale **null**

L'operatore **new**

- L'operatore **new** serve per creare un nuovo oggetto:
 - è seguito dal nome della classe di cui si vuole creare un'istanza e da opportuni parametri, che saranno usati (da un opportuno costruttore) tipicamente per inizializzare lo stato dell'oggetto appena creato

EsprCrealistanza ::= new IdQual (Args?)

*Args ::= Espressione (, Espressione)**

nota che (e) sono simboli terminali, mentre (e) sono parentesi usate per raggruppare gli elementi sintattici a cui applicare *.

L'assegnamento

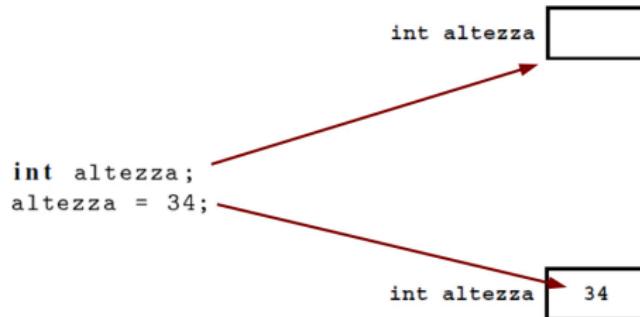
- L'**assegnamento** è un'istruzione elementare
 - modifica il contenuto di una determinata variabile con il valore risultante dalla valutazione di un'espressione

EsprAsgn ::= IdQual = Espressione

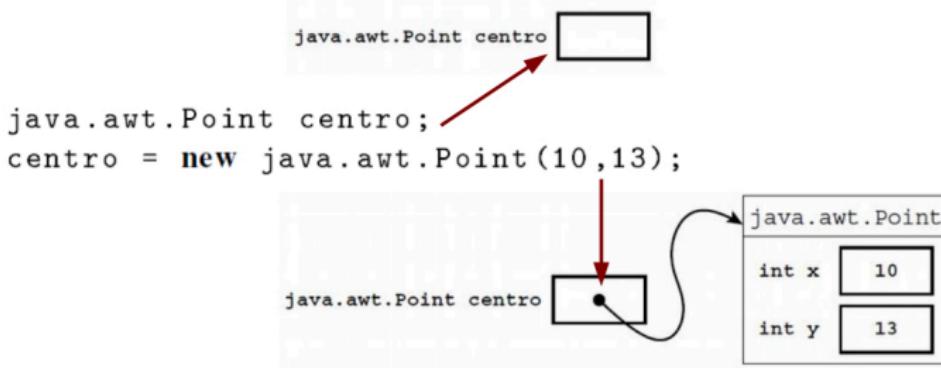
- Java è **type-safe**, nel senso di garantire che:
 - L'espressione a sinistra del simbolo di assegnamento deve denotare una variabile
 - L'espressione a destra del simbolo di assegnamento deve essere di tipo compatibile a quello della variabile

Assegnamento: tipo primitivo e tipo riferimento

- Assegnamento a tipo primitivo: la variabile contiene il valore



- Assegnamento a tipo riferimento: la variabile contiene un riferimento all'oggetto



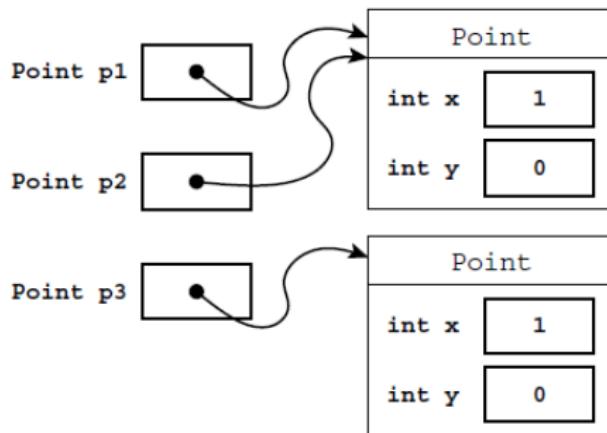
Espressioni con variabili

- Il nome di una variabile permette di far riferimento al suo valore all'interno di un'espressione
 - es: `area = base * altezza;`
 - es: `costo = (1 - sconto) * costo;`
- Ma attenzione
 - un assegnamento tra variabili di tipo primitivo causa una copia del valore
 - un assegnamento tra variabili di tipo riferimento causa una copia del riferimento (e la condivisione dell'oggetto)

Condivisione/Aliasing di oggetti

- Più variabili possono riferire allo stesso oggetto, essere **alias** per l'oggetto

```
Point p1 = new Point(1,0);
Point p2 = p1;
Point p3 = new Point(1,0);
```



Le costanti

- Una **costante** si dichiara esattamente come una variabile, ma premettendo il modificatore **final**
 - a differenza di una variabile, il valore di una costante viene fissato una sola volta e mai più cambiato
- Per convenzione, i nomi delle costanti sono scritti tutti con lettere maiuscole
 - se composti da più parole, si usa `_` per separarle

```
final int BOTTIGLIE_PER_CONFEZIONE = 6;  
final double PI = 3.141592  
final int SPEDD_LIMIT = 50 // Km/ora  
final char SEPARATORE = ':';
```

Confronto tra variabili per identità

- Il confronto per identità si basa sull'operatore ==
 - da non confondersi con l'operatore di assegnamento =
 - l'operatore == si applica a due espressioni di tipo compatibile, restituisce un booleano (true o false) ed è commutativo (l'esito non dipende dall'ordine degli argomenti)
- Applicato a due variabili di tipo primitivo, controlla se i rispettivi valori coincidono
- Applicato a due variabili di tipo riferimento controlla se le due variabili riferiscono lo stesso oggetto

```
Point p1 = new Point(1,0);
Point p2 = p1;
Point p3 = new Point(1,0);

( p1 == p2 )    ==> true
( p1 == p3 )    ==> false
```

Confronto tra variabili per uguaglianza

- Per tipi riferimento, un confronto spesso più utile è quello per **uguaglianza**, cioè un confronto che tiene conto della semantica dell'oggetto.
 - viene effettuato mediante il metodo `equals()` che vedremo in seguito
 - l'esito può essere positivo anche quando le variabili memorizzano riferimenti diversi, purché gli oggetti riferiti siano ritenuti equivalenti.
- Es: due oggetti distinti di tipo `java.awt.Point` sono ritenuti uguali se rappresentano punti del piano cartesiano con le medesime coordinate.

```
Point p1 = new Point(1,0);
Point p2 = p1;
Point p3 = new Point(1,0);

p1.equals(p3)    ==> true
( p1 == p3 )    ==> false
```

I package (1)

- Le classi Java sono organizzate in **packages**, che formano una gerarchia (come le directory)
- Il nome completo di un package contiene i nomi dei package che lo contengono separati da ":"
 - Es: `java.util.concurrent` è il nome completo di `concurrent`, sottopackage di `util`, sottopackage di `java`
- In una classe si dichiara il package cui appartiene con la direttiva **package** all'inizio del file seguita dal nome del package
 - La classe `Input` contiene la direttiva **package** `jbook.util;`
- Se manca la direttiva, la classe appartiene a un **package senza nome (unnamed o default in eclipse)**, associato alla directory in cui si trova

I package (2)

- Le classi all'interno di un package devono avere nomi diversi
- Nella classe A si può far riferimento a una classe B con il suo nome semplice se
 - B è nello stesso package di A
 - B è nel package `java.lang`
 - la classe A è preceduta da una direttiva **import** della classe B con il suo nome completo
- Altrimenti occorre usare il nome completo
 - Es: se nella classe Hello cancelliamo la direttiva **import** alla linea 5., dobbiamo sostituire alla linea 11.

```
Input.readString();
```

con

```
jbook.util.Input.readString();
```

Corso: Paradigmi di Programmazione

Paola Giannini

Alcuni tipi fondamentali

Gli otto tipi primitivi di Java

I tipi primitivi

Nome	Tipo di valore	Memoria usata	Range di valori
byte	intero	8 bit = 1 byte	-128 : +127
short		16 bit = 2 byte	-32.768 : +32.767
int		32 bit = 4 byte	-2.147.483.648 : +2.147.483.647
long		64 bit = 8 byte	-9.223.372.036.854.775.808 : +9.223.374.036.854.775.808
float	floating point	32 bit = 4 byte	+/- 3,4028... x 10 ³⁸ : +/- 1,4023... x 0-45
double		64 bit = 8 byte	+/- 1,767... x 10 ³⁰⁸ : +/- 4,940... x 0-324
char	singolo carattere	16 bit = 2 byte	Tutti i caratteri Unicode
boolean	true o false	1 bit	true o false

Costanti letterali

- Le **costanti letterali** sono sequenze di caratteri riservate
- Servono per indicare valori di tipi primitivi direttamente nel codice
 - i letterali numerici iniziano sempre con una cifra
 - i letterali di tipo char iniziano e finiscono con apice singolo
 - i letterali di tipo **boolean** sono solo **true** e **false**

Note sui tipi primitivi

- Rappresentazione indipendente dalla piattaforma
 - Codifiche in complemento a 2 (per i tipi interi) e in virgola mobile (per **float** e **double**)
 - Codifica UNICODE: <http://www.unicode.org> per i caratteri
- Ad ogni tipo primitivo corrisponde una cosiddetta **classe involucro** (**wrapper** o classe contenitore)

Tipo primitivo	Classe involucro
byte	Byte
short	Short
int	Integer
long	Long

Tipo primitivo	Classe involucro
float	Float
double	Double
char	Character
boolean	Boolean

Le classi involucro di Java

- Le classi involucro favoriscono il trattamento omogeneo di valori di tipo primitivo e di oggetti
 - le istanze di una classe involucro sono usate per encapsulare valori primitivi all'interno di un oggetto
 - i metodi della classe definiscono le operazioni ammissibili su tali valori
- La conversione da un tipo di dati primitivo verso la corrispondente classe involucro e viceversa è automatica (**auto-boxing** e **auto-unboxing**)

Costruttori classi involucro e `toString()`

- Ogni classe involucro è dotata di un **costruttore** che riceve come argomento il valore da incapsulare
- Il metodo `toString()` restituisce una **descrizione testuale** (stringa) del valore incapsulato dall'oggetto
- Gli oggetti delle classi involucro sono **immutabili** (come i tipi primitivi)!
- Esempi

```
Integer costo = new Integer(320);
Boolean tr = new Boolean(true);
String str1 = costo.toString(); //assegna "320" a str
String str2 = tr.toString();    //assegna "true" a str
```

I booleani

- Il tipo **boolean** è scorrelato dai tipi numerici (cioè un intero non si può usare come espressione di test in condizionali e/o cicli)
- l'insieme dei suoi valori comprende solo: vero e falso
- le uniche costanti letterali sono **true** e **false**.

I caratteri

- Il tipo **char** usa 16 bit per ogni carattere
 - codifica UTF-16 di UNICODE
 - include tutti i simboli degli alfabeti occidentali
 - i primi 127 codici coincidono con la codifica ASCII
- Utilizzabile come tipo numerico
 - valori senza segno: da 0 a 65535
- Le costanti letterali **char**
 - Apici singoli che racchiudono
 - un qualsiasi carattere singolo ASCII diverso da separatori di linea, apice singolo e backslash
 - o una sequenza di escape
 - sequenze di escape mnemoniche (es: '\n', '\t', '\')
 - sequenze di escape esplicite: '\uXXXX' (dove XXXX è un codice UNICODE esadecimale di 4 cifre)

Alcune costanti e metodi della classe Character

- SIZE: numero di bit a disposizione
- MIN_VALUE: minimo valore rappresentabile
- MAX_VALUE: massimo valore rappresentabile
- isWhitespace(Character c): testa se il carattere è di spaziatura
- isLetter(Character c): testa se il carattere è alfabetico
- toUpperCase(Character c): conversione a maiuscola

Ad esempio

```
Character c1= new Character('a');  
System.out.println(Character.isLetter(c1)); // stampa true  
System.out.println(Character.toUpperCase(c1)); // stampa A
```

I tipi numerici

- Valori in un certo intervallo
 - Numeri interi (positivi e negativi)
es: 0, -1, 100 o 3200
 - Numeri (positivi e negativi) con parti decimali
es: -9.99, 3.14159 o 10.0
- Valori grandi a piacere o con precisione arbitraria
 - `java.Math.BigInteger`
 - `java.Math.BigDecimal`

- Dati n bit, la rappresentazione in complemento a due fissa l'intervallo dei valori rappresentabili a cavallo dello 0 :
 - Numeri interi (positivi e negativi)
es: 0, -1, 100 o 3200
 - da $-2^{(n-1)}$ a $2^{(n-1)} - 1$ (in totale, 2^n valori rappresentabili)
- Alcune costanti delle classi involucro
 - SIZE: numero di bit a disposizione
 - MIN_VALUE: minimo valore rappresentabile
 - MAX_VALUE: massimo valore rappresentabile

Il trabocco (overflow)

- In Java l'aritmetica intera è modulare
 - l'eventuale **trabocco (overflow)** prodotto da un'operazione che produce un valore non rappresentabile non viene segnalato

```
long m=Long.MAX_VALUE;  
System.out.println(m+"*"+m+" = "+(m*m));
```

il programma stampa

9223372036854775807*9223372036854775807 = 1

mentre il risultato è

85070591730234615847396907784232501249

java.Math.BigInteger

- Le classi involucro hanno gli stessi limiti dei tipi primitivi rispetto agli intervalli dei valori rappresentabili
- La classe BigInteger del package java.Math permette di rappresentare e manipolare valori interi senza limiti di dimensione
- Le operazioni aritmetiche sono realizzate da opportuni metodi (es, multiply(BigInteger))

```
BigInteger m=new BigInteger("9223372036854775807");
System.out.println(m+"*"+m+" = "+(m.multiply(m)));
```

- il programma stampa

```
9223372036854775807*9223372036854775807 =
85070591730234615847396907784232501249
```

Le costanti letterali intere

- Le costanti letterali intere possono essere espresse in notazione decimale, ottale o esadecimale
 - per default: nell'intervallo dei valori **int**
 - se terminano per l o L: nell'intervallo dei valori **long**

LettIntero ::= *LettDOE* (l | L)?

LettDOE ::= *LettDec* | *LettOtt* | *LettEsa*

LettDec ::= 0 | *DigitNonZero* *Digit**^{*}

LettOtt ::= 0 *DigitOtt*⁺

LettEsa ::= 0 (x | X) *DigitEsa*⁺

Digit ::= 0 | *DigitNonZero*

DigitNonZero ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

DigitOtt ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

DigitEsa ::= *Digit* | a | b | c | d | e | f | A | B | C | D | E | F

- quale è il valore di x e y che seguono?

```
int x=011;      int y=0x11;
```

Valori in virgola mobile

- La rappresentazione in virgola mobile è basata sulla **notazione esponenziale**,

$$m \times b^e$$

dove

- b è la **base** della rappresentazione
 - m è la **mantissa**
 - e è l'**esponente**
 - il **segno** è + o -
 - es: $64.5 = 0.645 \times 10^2 = 645 \times 10^{-1}$
- **Forma normalizzata** fissa la posizione della virgola nella mantissa
 - **Standard IEEE 754**
 - I valori rappresentabili sono un sottoinsieme (finito e simmetrico rispetto allo 0) dei numeri razionali rappresentabili
 - **Precisione singola** (float)
 - **Precisione doppia** (double)

Errori di arrotondamento

- Il trabocco è trascurabile per valori **double**
- La rappresentazione esatta potrebbe non essere possibile
 - passaggio di base (a numeri decimali con rappresentazione finita possono corrispondere numeri binari con infinite cifre)
 - conversione tra valori interi e valori in virgola mobile
 - operazioni di calcolo
 - l'errore può propagarsi (e ampliarsi) nei calcoli successivi
- Il valore viene sempre approssimato col valore rappresentabile più vicino (arrotondamento)

```
System.out.println("2.1-2 = " + (2.1-2));
```

stampa

2.1-2 = 0.1000000000000009

Le costanti letterali in virgola mobile

- Esprimibili in notazione decimale o esadecimale

- una parte intera, seguita da . e da una parte frazionaria
- optionalmente, un esponente
- optionalmente, la precisione (singola o doppia)

LettVirgolaMobile ::= *LettVMDec* | *LettVMEsa*

LettVMDec ::= *Digit*⁺ . *Digit*^{*} *Esponente*? *Precisione*?
| . *Digit*⁺ *Esponente*? *Precisione*?
| *Digit*⁺ *Esponente* *Precisione*?
| *Digit*⁺ *Esponente*? *Precisione*

Esponente ::= (e | E) *Segno*? *Digit*⁺

Segno ::= + | -

Precisione ::= f | F | d | D

Alcune costanti delle classi involucro

Alcune costanti delle classi involucro `Float` e `Double`

- `SIZE`: numero di bit a disposizione
- `MIN_VALUE`: minimo valore rappresentabile
- `MAX_VALUE`: massimo valore rappresentabile
- `POSITIVE_INFINITY`: si ottiene dividendo un valore positivo per `0.0` (solo nelle classi `Float` e `Double`)
- `NEGATIVE_INFINITY`: si ottiene dividendo un valore negativo per `0.0` (solo nelle classi `Float` e `Double`)
- `NaN`: si ottiene dividendo `0` per `0.0` (solo nelle classi `Float` e `Double`)

La conversione di tipo

- Tipi numerici diversi corrispondono a intervalli diversi di rappresentazione
- Una conversione accidentale di tipo può causare una **perdita di informazione** non voluta
- Il compilatore effettua controlli mirati
 - es. il tipo dell'espressione a destra del simbolo di assegnamento deve essere **compatibile verso** il tipo della variabile a sinistra del simbolo di assegnamento
- Il programmatore può forzare la **conversione**

Conversione (cast) automatico

- Il compilatore ritiene un tipo compatibile verso un altro se la conversione NON provoca perdita di informazione
 - in questo caso la conversione è automatica e trasparente al programmatore (**cast automatico**)
 - nel caso di tipi numerici, si può passare da un tipo con minore precisione a un tipo con maggiore precisione
 - (un altro caso è dovuto al meccanismo di ereditarietà: ogni classe involucro di valori numerici è compatibile verso la classe `Number` e tutte le classi sono compatibili verso `Object`)
- La relazione di **compatibilità** indotta dal cast automatico è transitiva e riflessiva, ma non simmetrica

Cast automatici per i tipi primitivi

I caratteri possono essere usati come interi (valori senza segno: da 0 a 65535)



Cast esplicito

- Il **cast esplicito** è una semplice primitiva che agisce su un'espressione per modificarne il tipo $Espressione ::= (\text{Tipo}) \text{Espressione}$
- i seguenti assegnamenti sono tutti leciti

```
double d = 3.5;
int i = (int) d; // Faccio il cast esplicito
d = i;
Double dw = new Double(3.5);
```

mentre

```
int i = d;
```

non è corretto

Auto boxing e unboxing

- La conversione tra un tipo primitivo e la corrispondente classe involucro è automatica
 - auto-unboxing**: dalla classe involucro al tipo primitivo
 - integrato con il cast automatico (componibile transitivamente)
 - auto-boxing**: dal tipo primitivo alla classe involucro
 - applicato solo quando il tipo di arrivo è proprio la classe involucro corrispondente al tipo dell'espressione

```
Integer iInv = 30; //Auto-boxing: crea un oggetto (di tipo Integer)
                    //che contiene 30
int ii = iInv;      //Auto-unboxing: estrae dall'oggetto il valore
                    //tipo int
Integer e = ii + iInv; //Auto-unboxing di iInv, somma interi
                        //Auto-Boxing del risultato
Double dInv = 3.7; //Auto-boxing: crea un oggetto (di tipo Double)
                    //che contiene 3.7
double dD = dInv; //Auto-unboxing: estrae dall'oggetto il valore
                    //tipo double
short s=1;
Integer iS =s; //Errore (non si converte uno short a un Integer)
int id= dInv; //Errore (non si converte un Double a un int)
```

Le conversioni automatiche

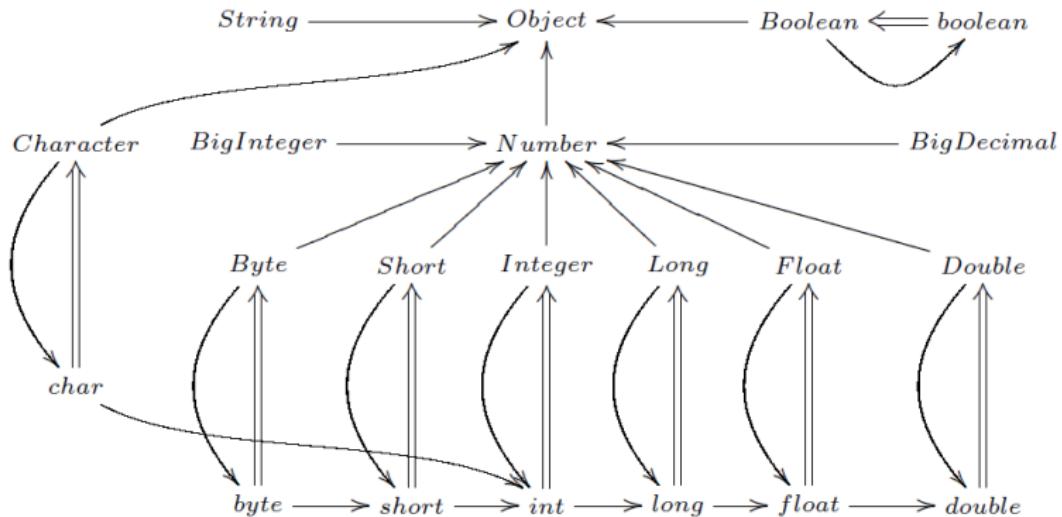


Figura 4.1: Cast automatico, auto-unboxing e auto-boxing

Le frecce singole → sono transitive, quelle doppie ⇒ non lo sono.

```
long l=new Byte((byte) 1); //OK  
Long la=new Byte((byte) 1); //ERRORE
```

Le sequenze

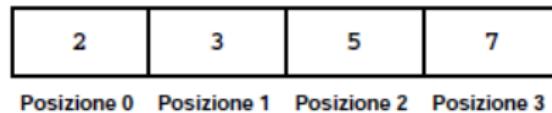
- Tutti i tipi visti riguardano valori elementari
- Spesso sorge la necessità di manipolare collezioni di dati omogenei
- Java offre diversi struttura dati per organizzare queste collezioni e agevolarne la gestione
 - strutture implementate direttamente nel linguaggio
 - strutture più sofisticate presenti nelle API

Gli array (1)

- Gli **array** sono presenti in molti linguaggi
 - si usano per rappresentare **sequenze** di elementi di uno stesso tipo (detto tipo base dell'array)
- Il numero n di elementi è detto **lunghezza**
 - determina le **posizioni ammissibili** (da 0 a $n - 1$)
 - non modificabile dinamicamente
- L'**accesso** è **diretto** (o random), cioè
 - ogni elemento è accessibile (in tempo costante) conoscendone la posizione
 - non occorre esaminare gli elementi che lo precedono

Gli array (2)

- Possiamo immaginare un array come costituito da una sequenza di variabili dello stesso tipo
 - nome omogeneo (distinte da un indice numerico)
 - il numero delle variabili è pari alla lunghezza dell'array
- Il numero n di elementi è detto **lunghezza**
- il tipo base può essere un qualsiasi tipo primitivo o riferimento (es: gli elementi possono essere interi oppure Boolean, oppure degli array a loro volta)



Dichiarazione di array

- Gli **array sono oggetti** (per la dichiarazione basta far seguire il tipo base da una coppia di parentesi quadre)
- Estendiamo la produzione per i **tipi riferimento** data nei lucidi (2) pagina 8.

Tipo ::= *TipoPrim* | *TipoRif*

IdQual ::= (*Id*.)* *Id*

TipoRif ::= *IdQual* | *Tipo* [] | ...

- Esempi

- Array **semplici** o **monodimensionali**

`String [] args; // array di stringhe`

`int [] a; // array di interi`

`Character [] c; // array di oggetti caratteri`

- Array **multidimensionali**

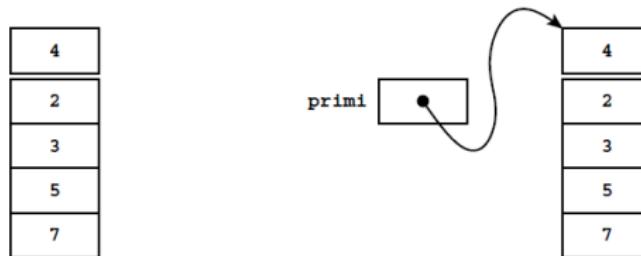
`char [] [] crossword; // due dimensioni`

`int [] [] [] m; // tre dimensioni`

La creazione di array

- Gli array sono oggetti e quindi si creano con **new**. La dichiarazione alloca solo la memoria per il riferimento!
 - la **new** alloca memoria per ogni elemento (la lunghezza deve essere nota)
 - e in più una cella per **memorizzare la lunghezza** che può essere ispezionata attraverso la variabile **length**, di tipo **int** (non modificabile)

oggetto array variabile **primi** che lo riferisce
(la prima cella contiene la lunghezza)



Creazione di array (1)

- La creazione di array ha una sintassi particolare, può
 - fissare la lunghezza per tutte le dimensioni, o
 - definire la lunghezza delle prime dimensioni dell'array e lasciare indefinite le ultime, o
 - definire lunghezza e contenuto mediante un'espressione di inizializzazione

Espressione ::= *EsprCreaArray* | *EsprIniArray* | ...

EsprCreaArray ::= new *Tipo*([*Espressione*])⁺ ([])^{*}
new *Tipo*([*Espressione*])⁺ *EsprIniArray*

EsprIniArray ::= { (*EsprIniVar* ,)^{*} *EsprIniVar* }

EsprIniVar ::= *Espressione* | *EsprIniArray* | ...

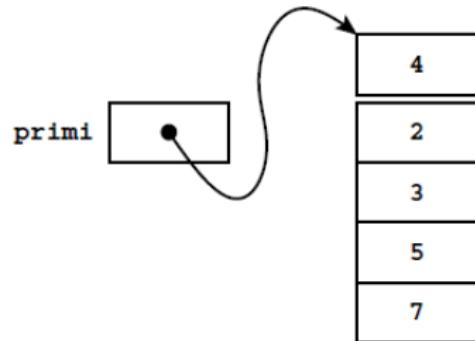
Creazione di array: inizializzazione di default

- Possiamo specificare la dimensione ma tralasciare gli elementi, che sono inizializzati automaticamente
 - con 0 se il tipo base è numerico
 - con **false** se il tipo base è booleano
 - con **null** se il tipo base è un tipo riferimento
- es:
 - **int[] primi = new int[4];**
associa alla variabile primi un array di lunghezza 4 i cui elementi sono tutti 0
 - **String[] sA=new String[3];**
associa alla variabile sA un array di lunghezza 3 i cui elementi sono tutti **null**

Creazione di array: espressione di inizializzazione

- `int[] primi;`

```
primi = new int[]{2,3,5,7};
```



- oppure

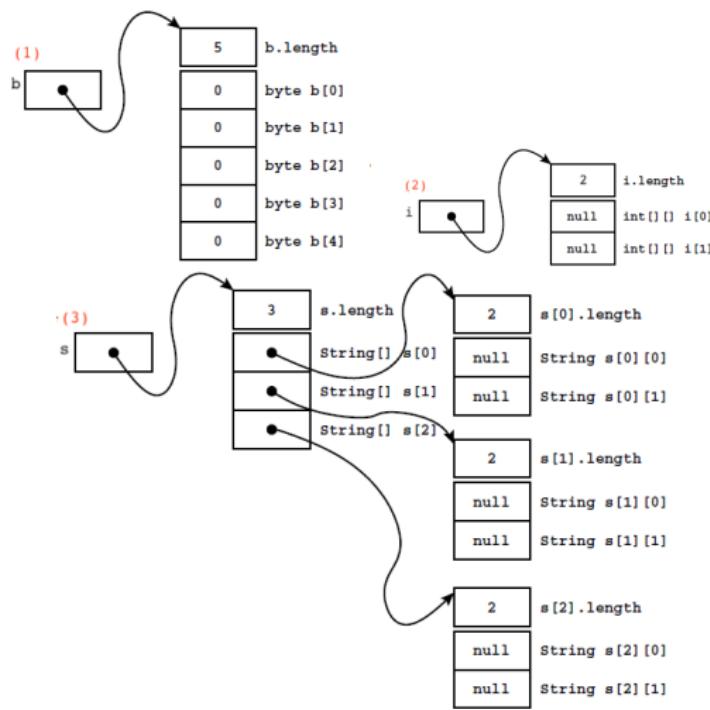
```
int[ ] primi = new int[]{2,3,5,7};
```

- ma non

```
int[ ] primi;  
primi = {2,3,5,7};
```

Creazione di array: esempi

- (1) `byte[] b = new byte[5];`
- (2) `int[][][] i = new int[2][][];`
- (3) `String[][] s = new String [3][2];`



Accesso agli elementi di un array e metodi utilità

- Per riferire i singoli elementi di un array si usa un'espressione, come in C, composta dal nome dell'array e da un indice intero racchiuso tra parentesi quadre.

Espressione ::= *AccediArray* ...

AccediArray ::= *Espressione* [*Espressione*]

- Non è possibile accedere attraverso il puntatore al primo elemento.
- es:
 - b[0] (byte)
 - s[0][1] (stringa)
 - s[2] (array di stringhe)
 - i[1] (array di array di **int**),
- Classe con metodi di utilità: `java.util.Arrays` e `System.arraycopy()` per fare copia.

Sequenze di caratteri: le stringhe

- Una **stringa** s è una sequenza di caratteri
 - la **lunghezza** di s è il numero n di caratteri che contiene
 - ogni carattere ha una **posizione** (o **indice**) che come per gli array, vanno da 0 a $n - 1$
- In Java le stringhe sono oggetti molto particolari
 - **immutabili**: il contenuto non è modificabile (questa proprietà è garantita dal fatto che `String` non mette a disposizione metodi per modificare lo stato delle istanze)
- `String` non è un tipo primitivo, ma possiede letterali
- possiamo creare istanze senza usare **new** (coi letterali)

Letterali di tipo String

- Sequenze di caratteri racchiuse tra apici doppi
- sono ammessi caratteri ASCII e sequenze di escape
- una sequenza particolare: la stringa vuota ""
 - da non confondersi con **null** (riferimento vacuo)

LettStringa ::= "Carattere"

esempio:

```
1. String s1="";
2. System.out.println("lunghezza di s1: " + s1.length());
3. String s2;
4. System.out.println("lunghezza di s2: " + s2.length());
5. String s3 = null;
6. System.out.println("lunghezza di s3: " + s3.length());
```

- 2. stampa lunghezza di s1: 0
- 4. Errore di compilazione
 - The local variable s2 may not have been initialized
- 6. Errore di esecuzione
 - Exception in thread "main"java.lang.NullPointerException

Alcuni operatori e metodi utili

- Concatenazione di stringhe: operatore +
- Lunghezza: metodo length()
- Costruttore String(**char**[]): da array di **char** a String (un array di caratteri NON è una stringa!)
- Confronto per uguaglianza: equals() stessi caratteri
 - variante: equalsIgnoreCase()
- Confronto per ordinamento lessicografico: compareTo()
 - variante: compareToIgnoreCase()

```
char[] cA = {'p','a','r','a','d','i','g','m','i'};  
String s4 = cA; // ERRORE  
String s5 = new String(cA); //OK  
String s6 = "Paradigm";  
String s7 = "paradigm";  
System.out.println("s5==s7? " + (s5==s7)); // s5==s7? false  
System.out.println("s5.equals(s7)? " + (s5.equals(s7))); // s5.equals(s7)? true  
System.out.println("s5.equals(s6)== " + s5.equals(s6)); //s5.equals(s6)==false  
System.out.println("s5.equalsIgnoreCase(s6)== " + s5.equalsIgnoreCase(s6));  
//s5.equalsIgnoreCase(s6)==true  
System.out.println("s6.compareTo(s5)== " + s6.compareTo(s5)); //s6.compareTo(s5)==-32  
System.out.println("s5.compareTo(s6)== " + s5.compareTo(s6)); //s5.compareTo(s6)==32  
System.out.println("s5.compareToIgnoreCase(s6)= " + s5.compareToIgnoreCase(s6));  
//s5.compareToIgnoreCase(s6)==0
```

Altri metodi utili e Conversioni di tipo

- Conversioni tipografiche e altre elaborazioni:
 - `toUpperCase()`, `toLowerCase()`, `trim()`
`replace(char trova, char rimpiazza)`
nota che rimpiazza NON modifica la stringa ma ne genera una nuova
- Accesso random: `charAt(int)`
- Conversione in array di caratteri: `toCharArray()`
- Estrazione di sottostringhe: `substring(int,int)`
- Scomposizione in array di stringhe: `split(String sep)`
- Ricerca di sottostringhe:
 - variante: `indexOf(String)`, `lastIndexOf(String)`, ecc.
- Conversione da tipo primitivo T var a stringa
 - concatenazione con la stringa vuota: `var + ""`
 - metodi `String.valueOf(T)`
- Conversione da tipo stringa String s a tipo primitivo
 - `Integer.parseInt(String)` ritorna (se la stringa è un letterale di intero) un `int`
 - `Double.parseDouble(String)` ritorna (se la stringa è un letterale di decimale) un `double`, ecc.... (usati nella classe `jbook.util.Input`)
- Conversione da oggetti obj a String : `obj.toString()`

La classe StringBuffer

- Simile a String, ma le sue istanze sono stringhe modificabili
 - sia nel contenuto
 - che nella lunghezza
- Il compilatore Java usa la classe StringBuffer per implementare l'operatore di concatenazione:
 - Metodi per inserire, cancellare, rimpiazzare uno o più caratteri del buffer

```
int n = 6;
String str = "L'intero n vale: " + n;
StringBuffer strB = new StringBuffer();

strB = strB.append("L'intero n vale: ");
System.out.println("strB prima dell'append"+ "\"" + strB + "\"");
strB = strB.append(n);
System.out.println("strB dopo l'append"+ "\"" + strB + "\"");
str =strB; //ERRORE non si puo' convertire StringBuffer a String
strB =str; //ERRORE non si puo' convertire String a StringBuffer
str = strB.toString(); //??
strB = new StringBuffer(str); // OK
```

I vettori: dichiarazione

- Sequenze di tipo omogeneo ma di lunghezza e contenuto variabili
- Classi `java.util.Vector` e `java.util.ArrayList`
 - sono **classi generiche**, parametriche rispetto al tipo base della sequenza,
 - il **tipo base** deve essere un **tipo riferimento**, (non può essere un tipo primitivo)
 - le funzionalità non dipendono dalla natura degli elementi
 - es: possiamo usare tipi riferimento quali
 - `ArrayList<String>`, `ArrayList<Integer>`
 - ma non `ArrayList<int>`

Dichiarazione vettori e metodi

```
TipoRif      ::=  IdQual ArgsDiTipo? | Tipo[ ]  
ArgsDiTipo   ::=  <ArgDiTipo ( ,ArgDiTipo )* >  
ArgDiTipo    ::=  TipoRif | ...
```

- Sequenze di tipo omogeneo ma di lunghezza e contenuto variabili

```
ArrayList<String> buffer = new ArrayList<String>();  
ArrayList<Integer> numeri = new ArrayList<Integer>();  
ArrayList<ArrayList<Integer>> sinonimi = new ArrayList<ArrayList<Integer>>();
```

- **Numero di elementi** presenti (dimensione): `size()`
- **Test** per sequenza vuota: `isEmpty()`
- **Accesso random**: `firstElement()`, `lastElement()`, `get(int)`
- **Sostituzione** di un elemento: `set(int pos, T el)`
- **Aggiunta** di un elemento (in fondo o in mezzo): `add(T el)`, `add(int pos, T el)`
- **Rimozione** di elementi (random, con ricerca, svuotamento): `remove(int pos)`,
`remove(Object el)`, `clear()`
- **Controllo di appartenenza**: `contains(Object el)`
- **Ricerca posizione**: `indexOf(Object el)`, `lastIndexOf(Object el)`, ecc...

Tipi Enumerativi

- Rappresentare insiemi ristretti di possibili valori
 - i semi di un mazzo di carte (cuori, quadri, fiori, picche),
 - il sesso di una persona (M, F),
 - i giorni della settimana (lunedì, ..., domenica),
 - i mesi dell'anno (gennaio, ..., dicembre),
 - le sigle di stati o di provincie,
 - ecc....
- Java offre una soluzione molto flessibile, potente e ben integrata col paradigma a oggetti

Tipi Enumerativi:dichiarazione

```
DichEnum ::= enum Id CorpoEnum
CorpoEnum ::= { CostEnums? }
CostEnums ::= CostEnum ( ,CostEnum )*
CostEnum ::= Id
```

- Le variabili del tipo enumerato potranno assumere soltanto i valori elencati nel definire il tipo enumerativo (e anche **null**).

```
enum Seme { CUORI, QUADRI, FIORI, PICCHE, }
enum Giorno { LUNEDI, MARTEDI, MERCOLEDI, GIOVEDI,
              VENERDI, SABATO, DOMENICA }
enum Mese { GENNAIO, FEBBRAIO, MARZO, APRILE,
            MAGGIO, GIUGNO, LUGLIO, AGOSTO,
            SETTEMBRE, OTTOBRE, NOVEMBRE, DICEMBRE }
```

```
Seme miaCarta;
miaCarta = Seme.CUORI;
Giorno domani = Giorno.MERCOLEDI;
Mese inizioAA = Mese.SETTEMBRE;
```

- Notate che gli elementi seguono la convenzione delle costanti!!

Corso: Paradigmi di Programmazione

Paola Giannini

Espressioni e Comandi

- Un' **espressione** è una porzione di codice
 - alla quale può essere **assegnato un tipo**
 - e la cui **valutazione**, se completata, produce **un valore di quel tipo**
- **Controllo dei tipi (type checking)**
 - fase di analisi di semantica statica
 - dedurre il tipo dell'espressione senza eseguire il codice
 - controllare che il tipo sia compatibile col contesto di uso (se necessario, col supporto di conversioni automatiche)

Le espressioni primarie

- Le **espressioni primarie** includono:
 - i letterali (già visti)
 - le invocazioni di metodi
 - l'accesso a variabili di oggetti e di classi
 - l'accesso a elementi di array (già visto)
 - le espressioni che creano array o istanze con **new** (parzialmente viste)
 - una qualsiasi espressione (che usa gli operatori Java) racchiusa tra parentesi
- Le **espressioni** sono costruite da altre espressioni con l'uso di operatori unari, binari, assegnamenti o costrutto condizionale.

EsprPri ::= *CostanteLetterale* | *AccediCampo* | *EsprNew*
| *AccediArray* | *InvocaMetodo* | (*Espressione*)

Espressione ::= *EsprUna* | *EsprBin* | *EsprAssgn* | *EsprCond*

AccediCampo ::= *EsprPri*.*Identificatore* | ...

AccediArray ::= (*IdQ* | *EsprPri*) [*Espressione*]

InvocaMetodo ::= *RifMetodo*(*Args*?)

RifMetodo ::=

Tipo e Valutazione di espressioni

- Per le espressioni il tipo deve essere determinabile a **tempo di compilazione**
 - così il compilatore garantisce che i valori assegnati alle variabili siano sempre compatibili verso il tipo dichiarato per quelle variabili
- Regole principali per assegnare il tipo
 - il letterale **null** ha un tipo speciale, chiamato **null**
 - i letterali booleani, caratteri, numerici e stringhe hanno l'ovvio tipo corrispondente
 - ogni variabile ha come tipo quello col quale è dichiarata
 - ogni operatore accetta solo argomenti di un certo tipo e produce solo risultati di un certo tipo
- La **valutazione di un'espressione** può
 - non terminare
 - produrre un errore (**lanciare un'eccezione**)
 - terminare regolarmente
 - producendo un valore
 - senza produrre alcun risultato (solo nel caso di invocazione a metodo con tipo del risultato **void**)

Overloading

- Lo stesso simbolo può denotare operazioni diverse
 - che operano su dati di tipo diverso
 - es: + denota somma e concatenazione di stringhe
 - es: la classe `java.lang.Math` contiene molti metodi con lo stesso nome, ad es. `max(int,int)`, `max(double,double)`,
`min(int,int)`, `min(double,double)`,
- Operatori e metodi con queste caratteristiche sono detti **sovraffunzionati** (*overloaded*)
 - Il tipo degli argomenti deve però determinare il tipo del risultato senza ambiguità

La promozione numerica binaria

- Può succedere che due operandi di un operatore numerico sovraccarico abbiano tipi compatibili verso tipi numerici diversi tra loro
 - quale versione dell'operatore deve essere considerata?
- La **promozione numerica binaria** risolve l'ambiguità
 - si applica agli operandi di alcuni operatori
 - produce solo risultati di tipo **int**, **long**, **float**, **double**
- Regole principali:
 - gli operandi che hanno tipo riferimento vengono convertiti tramite unboxing nei rispettivi tipi primitivi
 - se un operando ha tipo **double** anche l'altro viene convertito in **double**
 - se un operando ha tipo **float** anche l'altro viene convertito in **float**
 - se un operando ha tipo **long** anche l'altro viene convertito in **long**
 - altrimenti, entrambi vengono convertiti in **int**

Un Esempio

- Qual'è il tipo delle espressioni dalla riga 2. in poi? Ci sono conversioni ? Autoboxing e autounboxing ? espressioni

1. **int** x=2;
2. 6.3 * x
3. 6 * x
4. x / 3
5. x + "quattro"
6. Math.max(x,7)
7. Math.max(x,7.0)
8. x > 9.6
9. New Integer(x) + 7

Gli operatori sono come in C

- **aritmetici**: +, *,
- **incremento/decremento**: ++ e --
- **confronto**: <, <=,
- **uguaglianza**: ==, !=
- **logici**: ! (not)
 - &&, || (and e or) che hanno una valutazione **lazy** o **corto-circuitata** da sinistra a destra
- **bit-a-bit**: &, |, ^ (or esclusivo)
- **assegnamento semplice**: =
- **assegnamento composto**: $x \text{ op}=exp$ tale che
 - se x è una variabile di tipo T_1 ,
 - exp una espressione di tipo T_2 e
 - op un operatore che se applicato a valori di tipo T_1 e T_2 ritorna un valore compatibile con T_1 , allora
$$x \text{ op}=exp \Rightarrow x = (T_1) \text{ (} x \text{ op (} exp \text{))}$$
dove (T_1) è l'operatore di cast esplicito.

Operatori di cast e instanceof

- L'operatore di **cast esplicito** modifica a tempo di compilazione il tipo di una certa espressione
 - purchè questo sia compatibile verso quello del cast
- A tempo di esecuzione, il cast forza la conversione del valore, con potenziale perdita di precisione o generazione di errori

```
double d = 3.5;  
int i = (int) d; // i vale 3
```

- L'operatore **instanceof** si applica solo a espressioni (argomento di sinistra) che abbiano associato un tipo riferimento o **null**
- L'operando di destra deve essere il nome di un tipo riferimento
- Il valore restituito è **true** solo se l'operando di sinistra è un valore diverso da **null** e se il cast verso il tipo di destra è possibile.

Espressioni condizionali

- Sintassi:

$$\overbrace{\text{Espressione} \ ? \text{Espressione} : \text{Espressione}}^{\text{guardia}}$$

- la guardia deve avere tipo boolean
- Se T_1 è il tipo del secondo argomento e T_2 è il tipo del terzo argomento, il tipo T del risultato è il tipo più specifico al quale sia T_1 che T_2 possono essere convertiti
- La valutazione procede così: Si valuta la guardia
 - se **true**, il risultato è la valutazione del secondo argomento
 - se **false**, il risultato è la valutazione del terzo argomento
- Il valore da restituire viene convertito verso il tipo T determinato prima

```
int num=3;  
int den=2;  
float def =0;  
XX s7=den!=0?num/den:def;
```

XX può essere **int**? XX può essere **double**? e perchè

Operatori di uguaglianza e uguaglianza in virgola mobile

- Gli operatori == (uguale a) e != (diverso da) si applicano correttamente in tre casi:
 - entrambi gli operandi hanno un (tipo compatibile verso un) tipo numerico
 - entrambi gli operandi sono di tipo booleano
 - entrambi gli operandi hanno un tipo riferimento o **null**
- Il valore restituito è di tipo **boolean**
 - il confronto avviene per identità
- In assenza di effetti collaterali i due operatori sono commutativi
- È sempre garantito che $x!=y$ e $!(x==y)$ diano lo stesso risultato
- Abbiamo già detto che il **confronto per uguaglianza diretta tra due espressioni in virgola mobile è sconsigliabile**

perchè ci possono essere errori di approssimazione nella rappresentazione finita potrebbero rendere differenti valori idealmente uguali.

- Per confrontare si stabilisce una **soglia di accettabilità** e le espressioni sono confrontate a meno di tale soglia (la costante EPSILON)

`Math.abs(x-y) <= EPSILON`

invece di `x == y`

- se volete precisione alla 5 cifra decimale, che valore date a EPSILON?

Precedenza e associatività degli operatori

- In assenza di convenzioni, alcune espressioni potrebbero essere interpretate in maniera ambigua. Come si interpretano le seguenti espressioni?

$3 * 5 + 6 + 7 \quad i - j + +$

- La **precedenza** viene fissata (come in C) da una tabella. Le seguenti tabelle mostrano gli operatori, per precedenza decrescente **da sinistra a destra dall'alto in basso**. Notate che quelli a precedenza maggiore sono gli operatori unari.

- . -	dot notation	- * -	moltiplicazione	- == -	uguale a
- [_]	accesso elemento array	- / -	divisione o divisione tra interi	- != -	diverso da
- (_)	invocazione di metodo	- % -	resto della divisione intera	instanceof	appartenenza a un tipo
- ++	incremento postfisso	- + -	somma o concatenazione	- & -	AND bit-a-bit
- --	decremento postfisso	- - -	sottrazione	- ^ -	XOR bit-a-bit
++ -	incremento prefisso	- << -	shift aritmetico a sinistra	- -	OR bit-a-bit
-- -	decremento prefisso	- >> -	shift aritmetico a destra	- && -	congiunzione 'lazy'
! -	negazione booleana	- >>> -	shift logico a destra	- -	disgiunzione inclusiva 'lazy'
~ -	negazione bit-a-bit	- < -	minore di	- ? - : -	espressione condizionale
+ -	segno positivo (nessun effetto)	- <= -	minore o uguale a	- = -	assegnamento semplice
- -	inversione di segno	- > -	maggiore di	- op= -	assegnamento composto
(Tipo) -	cast esplicito	- >= -	maggiore o uguale a		(op uno tra *, /, %, +, -, <<, >>, >>>, &, ^,)
new -	creazione di oggetto				

- Tutto gli operatori, op, **associano a sinistra**, cioè

$$x \text{ op } y \text{ op } z \Rightarrow (x \text{ op } y) \text{ op } z$$

eccetto l'**espressione condizionale che associa a destra**,

$$x?y1:z1?y2:z2 \Rightarrow x?y1:(z1?y2:z2) \text{ NON } (x?y1:z1)?y2:z2$$

Dove trovare funzioni matematiche

- La classe **Math** raccoglie alcune costanti e metodi statici utili per il calcolo di funzioni matematiche
 - Approssimazione della costante di Nepero e base del logaritmo naturale: Math.E di tipo **double** (2.718281828459045)
 - Approssimazione di pi greco: Math.PI di tipo **double** (3.141592653589793)
 - Metodi vari per **int**, **long**, **float**, **double**: Math.abs(_), Math.min(_,_), Math.max(_,_)
- Funzioni trigonometriche: Math.sin(_), Math.cos(_), Math.tan(_),
- Generazione di numeri (pseudo)casuali: Math.random(_,_)
- Arrotondamento, potenze, radici, logaritmi: Math.floor(_), Math.pow(_,_), Math.sqrt(_), Math.log10(_),

I comandi in Java sono simili a quelli in C

Comando ::= Blocco | CmdEspr;
| *CmdIfElse* | *CmdSwitch*
| *CmdWhile* | *CmdDoWhile* | *CmdFor* | *CmdForEach*
| *CmdEtichettato* | *CmdBreak* | *CmdContinue*

- in nero i comandi elementari
- in verde i **comandi condizionali**: **if** e **if-else** come in C, ma la condizione DEVE essere di tipo **boolean**. Vedremo *CmdForEach* in seguito.
- in blu i **comandi iterativi**: **while**, **do-while** e **for** come in C , di nuovo la condizione DEVE essere di tipo **boolean**. Vedremo *CmdSwitch* in seguito.
- in rosso i **comandi che alterano il flusso dell'esecuzione**: come in C.

Blocco: dichiarazione e uso di variabili locali

Blocco ::= { *CmdInBlocco* }

CmdInBlocco ::= *Comando* | *CmdDichvarLoc* | ...

- Es: il corpo del metodo main della classe Hello è un blocco
- L'esecuzione di un blocco consiste nell'esecuzione delle istruzioni che contiene, nell'ordine dato
- L'**ambito di visibilità (scope)** di una variabile locale si estende dalla sua dichiarazione fino alla fine del blocco che la contiene
- Il compilatore controlla le seguenti regole di semantica statica:
 - Una variabile può essere usata solo nel suo ambito di visibilità
 - Non ci possono essere due variabili con nome uguale i cui ambiti di visibilità si sovrappongano
 - Un'istruzione che accede al valore di una variabile locale x deve essere preceduta da un assegnamento a x in ogni possibile flusso di esecuzione

Un programma errato

```
1. public class StampaProdottoErrato {
2.   public static void main(String[] args) {
3.     double a;
4.     {
5.       int a=5;
6.       int b;
7.       b=10;
8.     }
9.     System.out.println(b*a);
10.  }
11.}
```

- ① Lo scope della dichiarazione di a alla riga 5. si sovrappone a quello della dichiarazione di a alla riga 3.
- ② Lo scope della dichiarazione di b alla riga 7. finisce alla riga 8.
- ③ Alla riga 9. la variabile a è usata ma potrebbe essere non inizializzata!

Assegnamenti e comandi espressione

- Alcune espressioni possono essere usate come comandi, facendole seguire da ;

Comando ::= CmdEspr ; | ...

CmdEspr ::= EsprAss | EsprIncrDecr | InvocaMetodo | EsprCrealstanza

- L'espressione viene valutata e l'eventuale risultato viene scartato. Esempi:

- x=5;
- x++;
- Math.sqrt(4);
- new** String("Ciao!");

Il VECCHIO comando `switch`

- Il comando `switch` è molto utile con i tipi enumerativi, per eseguire azioni diverse per i diversi valori del tipo
- Es: definiamo il tipo enumerativo `Seme`

```
enum Seme { CUORI, QUADRI, FIORI, PICCHE }
```

- Vediamo come si può usare lo switch per leggere e/o stampare valori di questo tipo: lo stesso si può fare con qualunque tipo enumerativo

```
Seme seme;
char ch=Input.readChar("Quale seme? [c,q,f,p] ");
switch (ch) {
    case 'c':
        seme = Seme.CUORI;
        break;
    case 'p':
        seme = Seme.PICCHE;
        break;
    case 'q':
        seme = Seme.QUADRI;
        break;
    case 'f':
        seme = Seme.FIORI;
        break;
    default:
        seme = null;
}
```

Uso VECCHIO del comando `switch` con tipi enumerati

- Assegnare alla stringa `str` la stringa che corrisponde al seme

```
String str;
switch (seme) {
    case CUORI:
        str = "Cuori";
        break;
    case PICCHE:
        str = "Picche";
        break;
    case QUADRI:
        str = "Quadri";
        break;
    case FIORI:
        str = "Fiori";
        break;
    default:
        str = "Seme non riconosciuto";
}
```

Problemi con il comando `switch`

- Scrivere il **break** dopo ogni ramo è sia noioso che può portare ad errori!
- Cosa succede nel seguente caso?

```
String str;
Seme seme = Seme. PICCHE;
switch (seme) {
    case CUORI:
        str = "Cuori";
        break;
    case PICCHE:
        str = "Picche";
    case QUADRI:
        str = "Quadri";
        break;
    case FIORI:
        str = "Fiori";
        break;
    default:
        str = "Seme non riconosciuto";
}
```

Il NUOVO comando `switch`

- Dalla versione 14 sono stati introdotti sia un **nuovo comando** che una **nuova espressione** `switch`

```
String str;  
switch (seme) {  
    case CUORI ->str = "Cuori";  
    case PICCHE->str ="Picche";  
    case QUADRI->str = "Quadri";  
    case FIORI->str = "Fiori";  
    default ->str = "Seme non riconosciuto";  
}
```

```
String str1=  
switch (seme) {  
    case CUORI ->"Cuori";  
    case PICCHE->"Picche";  
    case QUADRI-> "Quadri";  
    case FIORI-> "Fiori";  
    default -> "Seme non riconosciuto";  
};
```

- Notate la differenza fra comando ed espressione e l'analogia con l'espressione condizionale.

Raggruppamento di clausole

- Si possono raggruppare clausole per le quali si vuole stesso comportamento
- Supponiamo che la stringa che vogliamo produrre ci dica se il seme è Rosso o Nero

```
String str;  
switch (seme) {  
    case CUORI, QUADRI ->str = "Rosso";  
    case PICCHE, FIORI->str ="Nero";  
    default ->str = "Seme non riconosciuto";  
}  
  
String str1=  
switch (seme) {  
    case CUORI, QUADRI -> "Rosso";  
    case PICCHE, FIORI-> "Nero";  
    default -> "Seme non riconosciuto";  
};
```

- Questo raggruppamento poteva essere fatto anche nella versione precedente del **switch**

Il comando `foreach` e l'iterazione su array

- Iterazione determinata su array (o vettori) è molto frequente. Es: stampa degli elementi di un array

```
int [] arr=new int [4];
for (int i =0; i< arr.length; i++) {
    System.out.println(" " + arr[i]);
}
```

- Java offre una sintassi alternativa, il `foreach`, per semplificare la scrittura

```
for (int next: arr) {
    System.out.println(" " + next);
}
```

Il comando `foreach`: sintassi

- La sintassi del `foreach` è la seguente:

$$CmdForEach ::= \text{for} \underbrace{(\ ModVarLoc? \ Tipo \ Id : Espressione)}_{intestazione} \underbrace{\text{Comando}}_{corpo}$$

- Espressione* nell'**intestazione** deve essere di tipo array (oppure di tipo Iterable: vedremo questo più avanti)
- Sia $T[]$ il tipo array di *Espressione*: allora T deve essere compatibile verso *Tipo*.
- $ModVarLoc?$ *Tipo* *Id* costituisce una dichiarazione di variabile locale, il cui ambito di visibilità comprende il **corpo**

Il comando `foreach`: esecuzione

- Un comando `foreach` viene eseguito così:

- Si valuta *Epressione* ottenendo un array
- Per ogni elemento dell'array si esegue il **corpo del `foreach`** dopo aver dichiarato la variabile *Tipo id* e avergli assegnato l'elemento considerato
- In altre parole,

```
for ( final? Tipo id: Epressione )
    Comando
```

viene eseguito come se fosse:

```
T[] newArr = Epressione;
for (int ind = 0; ind < newArr.length; ind++) {
    final? Tipo id = newArr[ind];
    Comando
}
```

Il comando `foreach`: pragmatica

- `foreach` ha una sintassi più compatta di `for`: non necessita di variable di controllo
- Dovete usarlo quando
 - si accede agli elementi dell'array in sola lettura
 - non serve la posizione di un elemento letto nell'array
 - si leggono in sequenza tutti gli elementi dell'array
- Es: somma, massimo, minimo, stampa degli elementi di un array

Corso: Paradigmi di Programmazione

Paola Giannini

Uno sguardo agli oggetti

Cosa sappiamo sugli oggetti

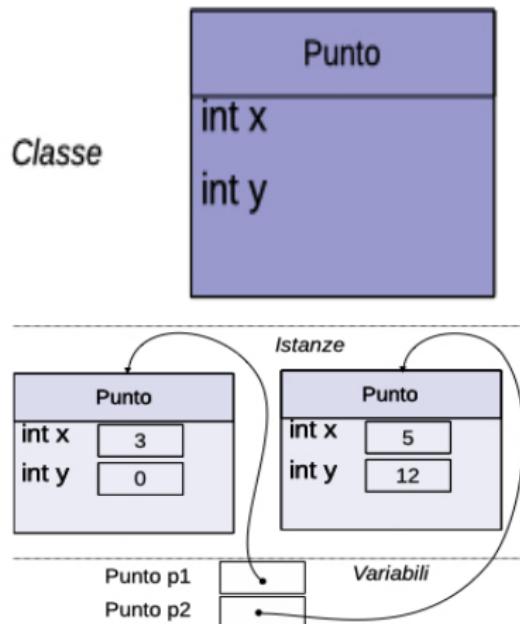
- Gli **oggetti** sono valori di **tipi riferimento**
 - le istruzioni di assegnamento, $p=q$, copiano un riferimento all'oggetto, non i valori contenuti nell'oggetto
 - le istruzioni di confronto, $p==q$, confrontano l'uguaglianza dei riferimenti, non dei valori contenuti nell'oggetto
- Gli oggetti encapsulano uno **stato** (es. oggetto di tipo Punto contiene le coordinate x e y del Punto)
 - gli operatori di selezione, $p.x$ consentono di riferire un particolare elemento dello stato interno
- Gli oggetti forniscono **operazioni**
 - la chiamata di metodo, $p.m()$ esegue l'operazione

Classi e istanze (1)

- Tutti gli oggetti dello stesso tipo sono **istanze** della stessa **classe**
- La classe definisce:
 - il **nome** e il **tipo** degli elementi dello stato di un oggetto (ma non il loro valore, che è diverso per ogni oggetto)
 - il **nome** e il **tipo** degli argomenti e dei risultati delle operazioni (**metodi**), nonchè il codice che deve essere eseguito quando le operazioni vengono invocate
 - Gli oggetti encapsulano uno **stato** (es. oggetto di tipo Punto contiene le coordinate x e y del Punto)
 - le **relazioni** della classe con altre classi e interfacce (argomento che vedremo in futuro)

Classi e istanze (2)

- Le classi fungono da “stampino” per gli oggetti
- Una classe definisce lo **schema** di come tutte le istanze di quella classe (oggetti) saranno fatte
- Le varie istanze di quella classe avranno tutte lo stesso schema, ma differenti valori
- A tempo di esecuzione, nell'heap ci sarà un (solo) oggetto che rappresenta la classe, detto **metaclasse** della classe.



Classi e istanze (3)

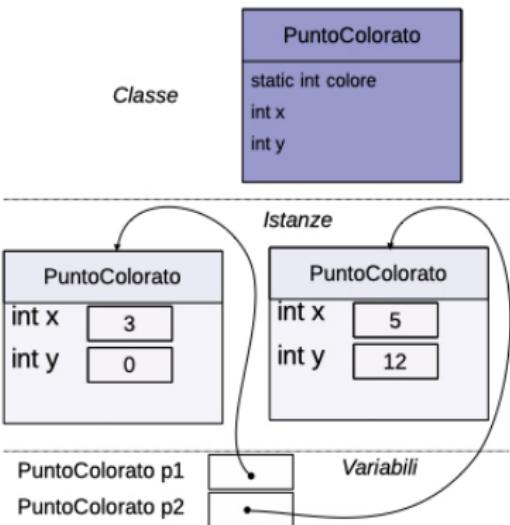
- Ogni classe è un **tipo** del linguaggio
- Java è un linguaggio **type-safe**
 - il compilatore può garantire, se la compilazione ha avuto successo, che ogni espressione ha il tipo “giusto” per l’uso che se ne fa
 - in particolare, ogni variabile che ha per tipo una classe conterrà sempre uno fra questi due:
 - un riferimento a un oggetto valido, istanza di quella classe
 - la costante speciale **null** che indica che la variabile non riferisce (in quel momento) alcun oggetto
 - nel primo caso, l’oggetto avrà sicuramente lo schema dato dalla classe

Elementi di classe e d'istanza (1)

- Abbiamo visto come ogni istanza (**oggetto**) contenga una propria copia degli elementi dello stato (**campi**), con un proprio valore
- È però possibile anche dichiarare che il valore per un certo elemento venga mantenuto direttamente dalla classe
- Si usa nella dichiarazione il qualificatore **static**

Esempio

- Tutte le istanze condividono colore
- `p1.colore=2` assegna 2 a colore nella classe
- `p2.colore` varrà anch'essa 2
- Le istanze non hanno un proprio campo colore
- Il campo colore sarà un **campo della metaclasse della classe**.



Elementi di classe e d'istanza: campi

- Parte della dichiarazione della classe PuntoColorato

```
public class PuntoColorato {  
    public static int colore; // variabile di classe  
    public int x,y; // variabile d'istanza  
    .....  
}
```

- Nella dichiarazione, le parti dello stato di classe sono distinte dalla presenza del qualificatore **static**
- Gli elementi di classe possono essere riferiti tramite istanze o tramite il nome della classe. Sono corretti
 - p1.colore
 - PuntoColorato.colore
- Gli elementi di istanza possono essere riferiti solo tramite le istanze.
 - p1.x e p1.y sono corretti
 - PuntoColorato.x e PuntoColorato.y NON sono corretti

Elementi di classe e d'istanza: metodi

- Anche i metodi possono essere di classe o d'istanza
 - i metodi di classe non possono riferire elementi di istanza, ma possono essere invocati sulla classe
 - i metodi di istanza possono riferire sia elementi di classe che elementi di istanza, ma devono essere invocati sulle istanze
- Si usa anche in questo caso il qualificatore **static**

```
public class Hello {  
    public static void main(String[] args) {  
        .....  
    }  
    .....  
}
```

- Il **main** di una classe eseguibile deve essere **static** perchè prima di iniziare le esecuzione, non ci sono istanze!

Elementi di classe e d'istanza: metodi

- Tutti gli oggetti della stessa classe avranno gli stessi metodi, con lo stesso codice
 - indipendentemente dal fatto che essi siano dichiarati **static** o meno
 - il risultato di un metodo di istanza applicato agli stessi parametri può essere diverso da un oggetto ad un altro perchè può dipendere dal valore dei suoi campi
- Riassumendo:
 - la classe ha un proprio stato (**variabili/campi/attributi di classe**)
 - la classe ha un proprio comportamento (**metodi**)
 - le istanze hanno un proprio stato (**variabili/campi/attributi di istanza**)

Dichiarazione di classi

- La dichiarazione delle classi ha una sintassi abbastanza complessa
- Vediamo dapprima il caso più semplice
 - caratteristiche più avanzate saranno introdotte gradualmente nelle prossime lezioni
- Fondamentalmente, la dichiarazione di una classe specifica quali sono i
 - suoi campi (variabili) e
 - metodi e
 - costruttori

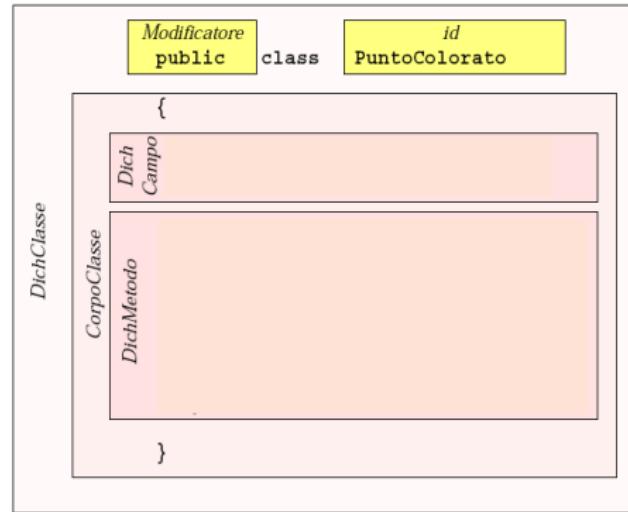
Dichiarazione di classi: il contenitore

DichClasse ::= *Modificatore** **class** *Id* *CorpoClasse*

CorpoClasse ::= { *DichInClasse** }

DichInClasse ::= *DichCampo* | *DichCostruttore* | *DichMetodo* | ...

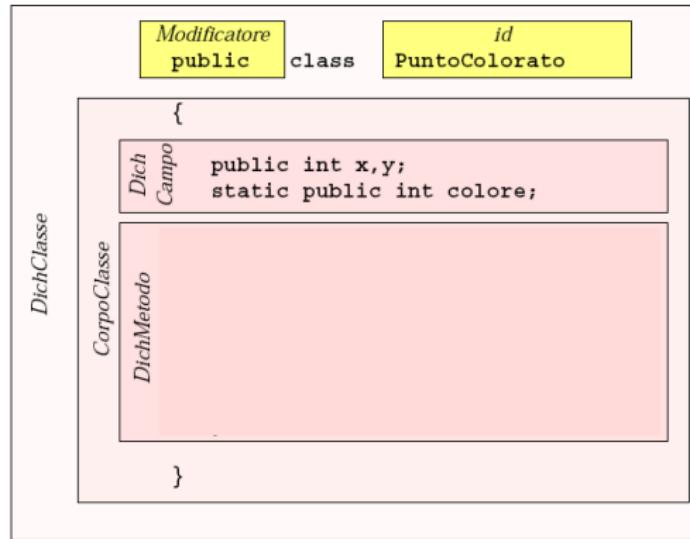
- Nell'esempio, vediamo come i vari elementi della dichiarazione di PuntoColorato corrispondano alle produzioni della dichiarazione di classe



Dichiarazione di classi: i campi

DichCampo ::= *Modificatore** *Tipo* *DichVars*

Modificatore ::= **public** | **private** | **protected** | **final** | **static** | ...



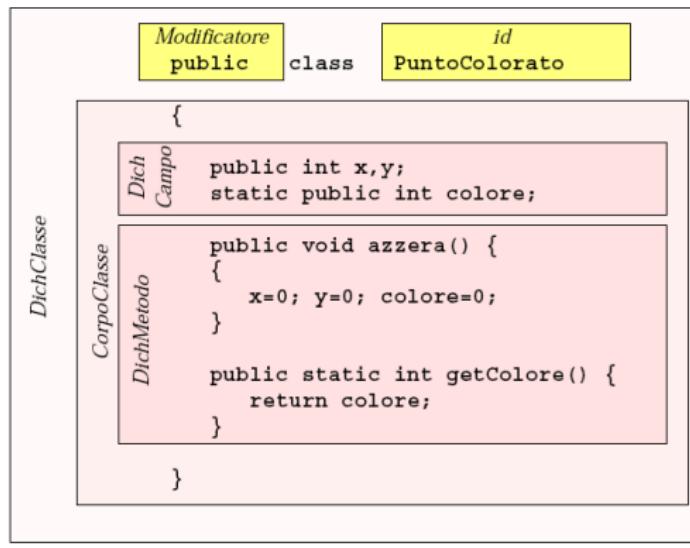
Dichiarazione di classi: i metodi

DichMetodo ::= *Modificatore** *Tipo* *Id* (*Params*?) *CorpoMetodo*

CorpoMetodo ::= *Blocco*

Params ::= *Param* (, *Param*)*

Param ::= *ModVarLoc*? *Tipo* *Id*



Dichiarazione di classi: i costruttori

DichCostruttore ::= Modificatore Id (Params) CorpoMetodo*

- La dichiarazione di un costruttore è analoga a quella di un metodo, con due eccezioni:
 - manca il tipo del valore restituito, e
 - *Id* è il nome della classe in cui è definito
- il costruttore è un particolare metodo che viene invocato automaticamente nel momento in cui si crea una nuova istanza (con **new**(_))
- il suo scopo è di inizializzare i campi dell'istanza
- non restituisce risultati

Dichiarazione di classi: un esempio completo

```
public class PuntoColorato {  
    static public int colore; //variabile di classe (statiche)  
    public int x, y; //variabili d'istanza  
  
    public PuntoColorato() { //costruttore  
        x = 0;  
        y = 0; //inizializza le variabili d'istanza  
    }  
  
    public void azzera() { //metodo d'istanza  
        x = 0;  
        y = 0;  
        colore = 0; //puo' accedere a variabili statiche  
    }  
  
    public static int getColore() { //metodo statico  
        return colore; //puo' accedere solo a variabili statiche  
    }  
  
    public static void setColore(int nuovoColore) { //metodo statico  
        // (con parametri)  
        colore = nuovoColore; //puo' accedere solo a variabili statiche  
    }  
}
```

Creazione di oggetti

- sintassi semplificata dell'espressione **new**

EsprCrealstanza ::= new IdQ (Args?)

*Args ::= Espressione (, Espressione)**

- *IdQ* è il nome della classe di cui si vuole creare un'istanza (un identificatore qualificato). Es. abbiamo visto

p1= **new** java.awt.Point(1,0)

- Il risultato dell'operatore **new**, è un riferimento al nuovo oggetto appena creato
- Il riferimento restituito da **new** può essere assegnato a una variabile del tipo appropriato
 - l'assegnamento a una variabile è un caso frequente, ma assolutamente non l'unico possibile
 - Es: posso passare l'oggetto come argomento di un metodo

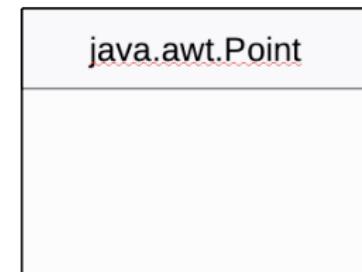
System.out.println(**new** Point(1,0));

Passi della creazione di un oggetto (1)

- La valutazione di un'espressione **new** è in realtà piuttosto complessa, e si articola in vari passi:
 - 1) si determina la classe che si vuole istanziare
 - nel caso di un nome completamente qualificato per *IdQ*, quello è già il nome della classe
 - altrimenti, per trovare il nome completo, si considerano le clausole **import**.
 - 2) se il codice della classe (il file **.class** relativo) non è già in memoria, lo si carica
 - se si carica la classe, viene creato un oggetto che rappresenta la classe, cioè la **metaclasse** della classe e vengono inizializzate le variabili statiche al loro valore di inizializzazione

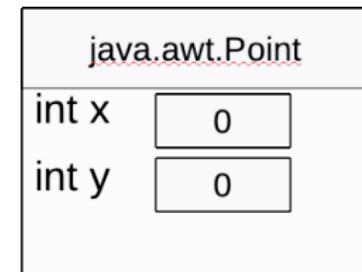
Passi della creazione di un oggetto (1): `new java.awt.Point(1,0)`

3) Viene quindi allocato nello heap un blocco di memoria grande a sufficienza per contenere gli elementi di istanza dello stato



4) La memoria allocata viene inizializzata

- tutte le variabili vengono inizializzate al loro valore di default: 0 per i valori numerici, **false** per i booleani e **null** i riferimenti
- vengono valutate le espressioni eventualmente presenti nelle inizializzazioni dei campi (generalmente non ci sono)

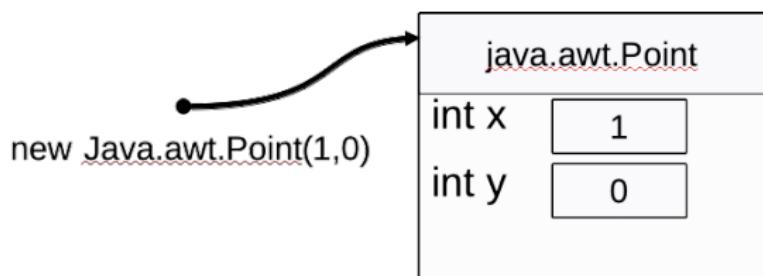
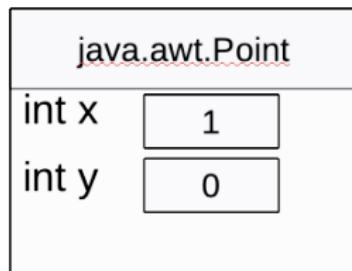


Passi della creazione di un oggetto (2): `new java.awt.Point(1,0)`

5) Viene ricercato fra i costruttori della classe quello "giusto", corrispondente ai parametri forniti nella `new`

- Il costruttore viene invocato, come un qualunque metodo, e riceve come argomenti i valori dei parametri attuali della `new`

6) Il valore restituito dall'espressione `new` è un riferimento al nuovo oggetto creato e inizializzato e il suo tipo è quello della classe menzionata nella `new`.



Distruzione di oggetti

- Visto che gli oggetti possono essere creati, è lecito chiedersi come vengano distrutti
- In Java esiste il **garbage collection** (raccolta della spazzatura) automatica
- La JVM identifica gli oggetti “spazzatura”
 - determinando il **reachable object graph (ROG)** delle variabili del programma (che sono nello **stack**), cioè gli oggetti raggiungibili dallo **stack**.
- Periodicamente vengono eliminati dalla memoria gli oggetti che non sono nel ROG e lo spazio reso disponibile allo heap per nuove allocazioni.
- L'effettiva cancellazione può avvenire qualche tempo dopo che l'ultimo riferimento a un oggetto è stato perso
 - per esempio, la cancellazione può essere rimandata finchè c'è altra memoria disponibile
- Al momento dell'effettiva cancellazione viene invocato, se presente, il metodo **public void finalize()**
- Nel corpo di **finalize()** possono essere liberate ulteriori risorse, eseguite stampe, ecc (all'uscita da **finalize()**, l'oggetto viene distrutto)

La classe Class (1)

- Esiste una particolare classe, di nome `Class`, le cui istanze sono rappresentazioni delle classi esistenti nel programma
- Si può ottenere un riferimento all'istanza di `Class` che rappresenta la classe di un oggetto o chiamando il metodo `getClass()` dell'oggetto:

`o.getClass()`

- Esiste anche una notazione per denotare le costanti letterali di tipo `Class`:

Tipo.class

- Gli oggetti della classe `Class` hanno, fra gli altri, un metodo `newInstance()` il cui comportamento è analogo a quello dell'operatore `new` (ma il tipo di ritorno è `Object`)
- Per esempio

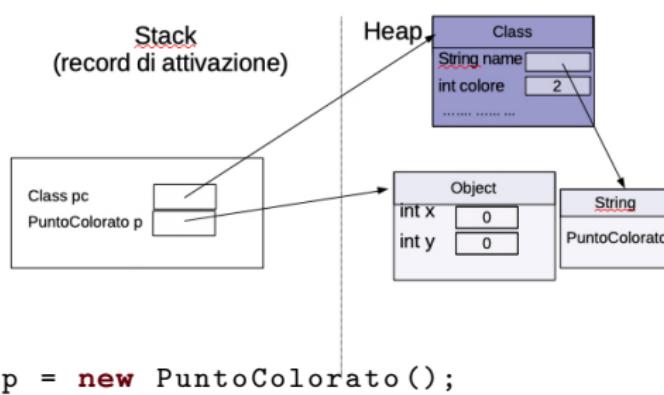
```
Class pc = PuntoColorato.class;
PuntoColorato p = (PuntoColorato) pc.newInstance();
```

è **quasi** equivalente a

```
PuntoColorato p = new PuntoColorato();
```

Stack e heap dell'esempio

```
Class pc = PuntoColorato.class;  
PuntoColorato p = (PuntoColorato) pc.newInstance();
```



La classe Class (2)

- La classe Class fornisce anche un metodo statico `forName()`
 - il metodo prende come argomento una stringa, e restituisce un riferimento all'istanza di Class che rappresenta la classe con il nome dato dalla stringa
- Esempio:

```
Class c = Class.forName(s);
Object o p = c.newInstance();
```

- In questo modo, è possibile operare su oggetti di tipo deciso **dinamicamente** a tempo d'esecuzione
 - usato per le architetture a **plug-in**

Corso: Paradigmi di Programmazione

Paola Giannini

I metodi

I metodi

- I **metodi** sono sequenze di istruzioni, dotate di parametri, legate a un oggetto o a una classe, e dotate di un nome mnemonico
 - come costrutto di **modularità**, svolgono un ruolo simile a quello di funzioni e procedure in altri linguaggi
 - come costrutto di **modellazione**, descrivono l'implementazione di operazioni offerte da una classe
- I metodi dispongono di un loro **stato locale**
 - costituito dalle **variabili locali**
- I metodi accettano **parametri** e restituiscono **risultati**

Dichiarazione di classi: i metodi

- I metodi sono sempre membri di una classe (vi ricordo la sintassi delle classi)

DichClasse ::= *Modificatore** **class** *Id* *CorpoClasse*

CorpoClasse ::= { *DichInClasse** }

DichInClasse ::= *DichCampo* | *DichCostruttore* | ***DichMetodo*** | ...

- L'ordine è tradizionale, ma non obbligatorio, **prima i campi, poi i costruttori, infine tutte le dichiarazioni di metodi**

Sintassi della dichiarazione dei metodi

DichMetodo ::= Modificatore ParamsDiTipo? Tipo Id (Params) CorpoMetodo*
CorpoMetodo ::= Blocco

- Il **corpo** del metodo è semplicemente un **blocco** può includere un'istruzione **return**, che ritorna al chiamante.
- In assenza del modificatore **static** il metodo è d'**istanza**, altrimenti è di **classe**.
- I parametri appaiono come una lista di dichiarazioni e si comportano nel corpo come variabili locali
- Esempio

```
1 public class Testo {  
2     int cnt = 0;  
3     public static void stampa(String testo, int x) {  
4         System.out.println(testo + " " + x);  
5     }  
6     public void stampacnt() {  
7         stampa("prova",3);  
8         System.out.println(cnt++);  
9     }  
10 }
```

Invocazione di metodi

- L'**invocazione** è l'operazione con cui si chiede l'esecuzione del blocco di codice associato a un metodo dato
- Il chiamante deve indicare:
 - su quale classe o oggetto è invocato il metodo
 - quale metodo della classe invocare
 - quali valori devono essere usati per i parametri attuali e occorre che i tipi dei parametri attuali siano compatibili con quelli dei parametri formali indicati nella dichiarazione
- Dal punto di vista sintattico, l'invocazione di metodo può essere
 - un'**espressione**, se il metodo restituisce un valore (ma può essere usata dove è necessario un comando facendola seguire da ;, punto e virgola)
 - un **comando**, se il tipo di ritorno del metodo, indicato nella dichiarazione, **void**.

Sintassi dell'invocazione di metodo

ExprPri ::= *InvocaMetodo* | ...

InvocaMetodo ::= *RifMetodo* (*Args*?)

Args ::= *Espressione* (, *Espressione*)^{*}

RifMetodo (senza generici)¹ può avere una fra queste tre forme:

RifMetodo ::= *Id* | (*Id*.)⁺ *Id* | *ExprPri*. *Id*

Se *RifMetodo* è

- ① *Id* allora *Id* è il nome di un metodo (statico o di istanza) della classe in cui si trova l'invocazione.
- ② *Id*₁. *Id*₂... *Id*_n allora *Id*₁... *Id*_{n-1} è il nome di una classe e *Id*_n è nome di un metodo statico della classe
- ③ *ExprPri*. *Id* allora *Id* è nome di un metodo di istanza della classe che è il tipo dell'espressione *ExprPri*.

¹vedremo i generici più avanti nella lezione

Esempi di invocazione di metodo delle tre forme

- Riprendiamo l'esempio di dichiarazione fatto in precedenza. Assumiamo che la classe sia definita nel **package testo**.

```
1 package testo;  
2  
3 public class Testo {  
4     int cnt = 0;  
5     public static void stampa(String testo, int x) {  
6         System.out.println(testo + " " + x);  
7     }  
8     public void stampacnt() {  
9         stampa("prova",3);  
10        System.out.println(cnt++);  
11    }  
12 }
```

- 1 chiama il metodo statico stampa() ma è valido solo all'**interno della classe Testo** (riga 9 del codice)

```
stampa("prova",3);
```

- 2 chiama il metodo statico stampa() passando i due parametri attuali "prova" e 3 (questa può essere in un metodo di una classe in un package diverso da testo)

```
testo.Testo.stampa("prova",3);
```

- 3 chiama stampacnt() sulla istanza di Testo contenuta nella quarta cella dell'array testi dell'oggetto riferito da doc.

```
public class Doc {  
    testo.Testo[] testi;  
  
    public void chiamataMet() {  
        Doc doc = new Doc();  
        doc.testi[3].stampacnt();  
    }  
}
```

Il comando `return`

- In seguito a un'invocazione, il corpo del metodo viene eseguito fino all'esecuzione di un'istruzione `return` oppure fino alla fine del corpo, quindi il controllo torna al chiamante
- `return` può essere seguito da un'espressione, che fornisce il valore di ritorno del metodo

Comando ::= *CmdReturn* | ...

CmdReturn ::= `return` *Espressione*? ;

- Se il metodo è ritorna `void`, allora il comando `return` è facoltativo, e se presente non deve comprendere un'espressione
- Se il metodo ritorna un altro tipo T, allora
 - 1 ci deve essere un comando `return` per ogni possibile flusso di esecuzione nel corpo, e `return` deve includere un'espressione;
 - 2 l'espressione deve essere convertibile al tipo T; il valore restituito sarà il risultato della conversione

Il meccanismo di chiamata

- L'invocazione di un metodo comporta l'esecuzione di un certo numero di passi:
 - ❶ valutazione del riferimento
 - ❷ valutazione degli argomenti
 - ❸ identificazione del corpo da eseguire
 - ❹ creazione del record di attivazione
 - ❺ esecuzione del corpo
 - ❻ rimozione del record di attivazione e ritorno
- Vediamo la valutazione attraverso un esempio (poi avete i passi in dettaglio).

Il record di attivazione dei metodi

- Simile a quello che avete visto per le funzioni, cioè contiene:
 - associazione fra parametri formali e valori degli argomenti (su cui è chiamato il metodo)
 - associazione fra variabile locali e loro valori
 - indicazione del punto del programma a cui ritornare
- I **metodi di istanza hanno un parametro隐式的**, l'oggetto su cui è chiamato il metodo, il **receiver** del metodo
 - questo oggetto è riferito attraverso la metavariabile **this**

Un esempio

```
public class Chiamato {  
    public static void statico(int x) {  
        System.out.println("metodo statico, x=" + x);  
    }  
    public double istanza(int a, int b) {  
        int s = a + b;  
        System.out.println("metodo di istanza, a=" + a + ", b=" + b);  
        return s / 2;  
    }  
}  
  
public class Chiamante {  
    public static void main(String args[]) {  
        int a = 3;  
        int b = 5;  
        double r = 0;  
        Chiamato.statico(a + b);  
        if (b > 1) {  
            Chiamato c = new Chiamato();  
            r = c.istanza(a + 1, 1);  
        }  
        System.out.println("finito, r=" + r);  
    }  
}
```

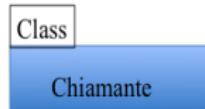
L'esecuzione inizia dal `main()` di `Chiamante`

Esecuzione (1)

```
1 public class Chiamante {  
2  
3  
4     public static void main(String[] args) {  
5         int a = 3;  
6         double r = 0;  
7         Chiamato.statico(a+7);  
8         if (a>2) {  
9             Chiamato c = new Chiamato();  
10            r = c.istanza(a+1);  
11        }  
12        System.out.println("finito r= " + r);  
13    }  
14  
15 }
```

- Dopo le dichiarazioni, lo stack conterrà il record di attivazione di main con args, a, r, con i rispettivi valori
- Nell'heap ci sarrà solo la metaclasses della classe Chiamante
- Quando arriviamo alla chiamata del metodo Chiamato.statico(a+7) viene caricata la classe Chiamato

a:	3
r:	0.0
args:	null



Esecuzione (2)

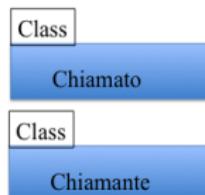
```
public class Chiamato {
    private int b = 1;

    public static void statico(int i) {
        System.out.println("metodo statico i= " + i);
    }

    public double istanza(int a) {
        int s = a + this.b; // oppure a + b
        System.out.println("metodo istanza a= " + a);
        return s / 2;
    }
}
```

- Con la chiamata, si passa ad eseguire il corpo di statico con parametro `i=10`
- Viene messo sullo stack il record di attivazione che contiene anche il punto di ritorno, il precedente record è “invisibile” e viene stampato
`metodo statico i=10`

i:	10
IR:	Riga 8 main
a:	3
r:	0.0
args:	null

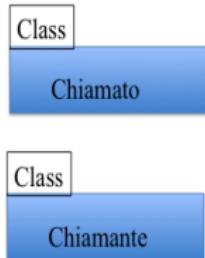


Esecuzione (3)

```
1 public class Chiamante {  
2  
3  
4     public static void main(String[] args) {  
5         int a = 3;  
6         double r = 0;  
7         Chiamato.statico(a+7);  
8         if (a>2) {  
9             Chiamato c = new Chiamato();  
10            r = c.istanza(a+1);  
11        }  
12        System.out.println("finito r= " + r);  
13    }  
14}  
15 }
```

- Al ritorno da `statico`, il record di attivazione creato per l'invocazione viene disallocato dallo stack
- l'ambiente precedente è di nuovo visibile

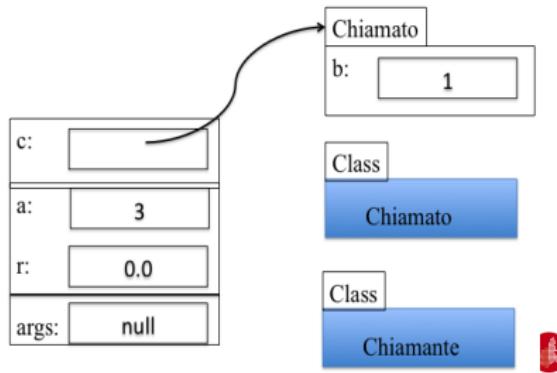
a:	3
r:	0.0
args:	null



Esecuzione (4)

```
1 public class Chiamante {  
2  
3  
4    public static void main(String[] args) {  
5        int a = 3;  
6        double r = 0;  
7        Chiamato.statico(a+7);  
8        if (a>2) {  
9            Chiamato c = new Chiamato();  
10           r = c.istanza(a+1);  
11       }  
12       System.out.println("finito r= " + r);  
13   }  
14 }  
15 }
```

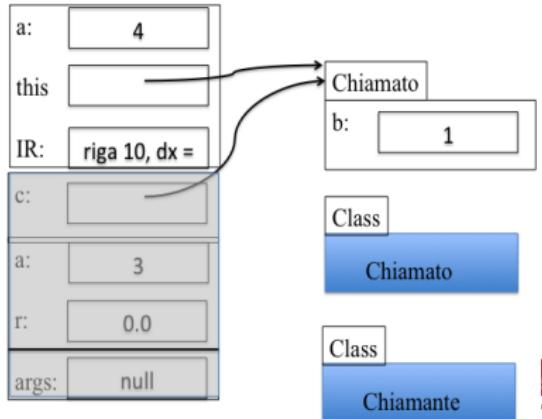
- Entrando nel ramo then dell'**if**, si crea uno scope distinto, ma le altre variabili sono ancora visibili
- si crea una istanza di Chiamato riferita da c
- per valutare l'assegnamento si deve prima valutare la chiamata del metodo `istanza(a+1)` su c
- per questo si deve valutare l'argomento `a+1` nell'ambiente corrente



Esecuzione (5)

```
public class Chiamato {  
    private int b = 1;  
  
    public static void statico(int i) {  
        System.out.println("metodo statico i= " + i);  
    }  
  
    public double istanza(int a) {  
        int s = a + this.b; // oppure a + b  
        System.out.println("metodo istanza a= " + a);  
        return s / 2;  
    }  
}
```

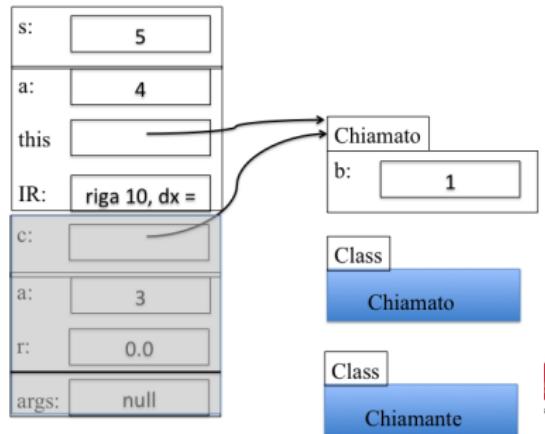
- Con la chiamata, si passa ad eseguire il corpo del metodo *istanza*
- Oltre al parametro *a* l'ambiente include **this**, legato all'oggetto riferito da *c*



Esecuzione (6)

```
public class Chiamato {  
    private int b = 1;  
  
    public static void statico(int i) {  
        System.out.println("metodo statico i= " + i);  
    }  
  
    public double istanza(int a) {  
        → int s = a + this.b; // oppure a + b  
        System.out.println("metodo istanza a= " + a);  
        return s / 2;  
    }  
}
```

- La dichiarazione di `s` introduce una variabile nel record di attivazione del metodo `istanza`
- Il valore di `a` letto è quello dell'ambiente al top della pila e quello di `b` è il campo dell'oggetto riferito da `this`
- Viene stampato
`metodo istanza a=4`
- si ritorna `s/2`, cioè?

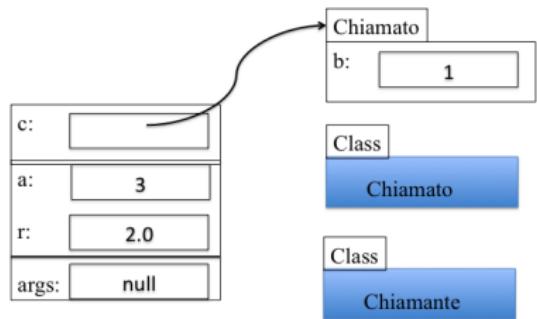


Esecuzione (7)

```
1 public class Chiamante {  
2  
3  
4     public static void main(String[] args) {  
5         int a = 3;  
6         double r = 0;  
7         Chiamato.statico(a+7);  
8         if (a>2) {  
9             Chiamato c = new Chiamato();  
10            r = c.istanza(a+1);  
11        }  
12    System.out.println("finito r= " + r);  
13 }  
14 }  
15 }
```



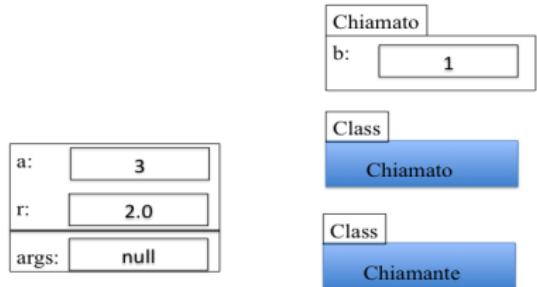
- Al ritorno, il record di attivazione di `istanza` viene disallocato dallo stack
- `r` assume il valore di ritorno (fornito dal `return`)



Esecuzione (8)

```
1 public class Chiamante {  
2  
3  
4    public static void main(String[] args) {  
5        int a = 3;  
6        double r = 0;  
7        Chiamato.statico(a+7);  
8        if (a>2) {  
9            Chiamato c = new Chiamato();  
10           r = c.istanza(a+1);  
11       }  
12       System.out.println("finito r= " + r);  
13   }  
14 }  
15 }
```

- All'uscita dal blocco, la variabile c non c'è più
- L'istanza di Chiamato creata dentro il blocco potrebbe essere disallocata perchè non più raggiungibile dal programma
- Viene stampato
`finito r=2.0`
- Con la fine dell'esecuzione del metodo main l'esecuzione del programma finisce.



1 - Valutazione del riferimento

- Se la parte prima del punto è un'espressione (caso 3 della definizione di *RifMetodo*), questa viene valutata; il risultato è l' oggetto su cui eseguire il metodo. Es.

```
doc.testi[3].stampacnt();
```

- Se invece la parte prima del punto è il nome qualificato di una classe (caso 2 della definizione di *RifMetodo*) questa è la metaclasses su cui eseguire il metodo. Es.

```
testo.Testo.stampa("prova",3);
```

non si deve fare la valutazione

- Se invece manca del tutto il punto (caso 1 della definizione di *RifMetodo*), allora il metodo viene invocato sulla stessa istanza o classe del codice che contiene l'invocazione. Es.

```
stampa("prova",3);
```

2 - Valutazione argomenti

- Vengono quindi valutate le espressioni fornite come argomenti nella chiamata.
 - in ordine da sinistra a destra
 - eventuali effetti collaterali della valutazione di una espressione sono visibili nella valutazione delle espressioni successive. Ad esempio

```
x=3;    y=4;  
ogg.met(x++, x++, x+y);
```

chiama il metodo `met` sull'oggetto riferito da `ogg` con argomenti ?
e dopo la chiamata quanto valgono `x` e `y`?

- I **risultati** della valutazione (non le espressioni!) diventano i parametri attuali della chiamata

3 - Identificazione del corpo

- Solitamente, è semplice identificare il corpo del metodo da chiamare, in base alla **firma (prototipo)**, cioè si cerca un metodo la cui intestazione coincida con la chiamata per:

- **nome** del metodo
- **numero e tipo** dei parametri (ma non il loro nome)

} firma

- Possono esserci situazioni più complesse (casi in cui per determinare il corpo c'è bisogno di ricerca a tempo di esecuzione)
- In ogni caso, un insieme di regole consente al compilatore di identificare il **corpo giusto** da eseguire.

4 - Creazione del record di attivazione

- Le istruzioni vengono eseguite in un ambiente dato dalle variabili locali (associate ai rispettivi valori)
- Ogni chiamata di metodo stabilisce un nuovo record di attivazione, in cui sono visibili i parametri formali e le variabili locali del metodo chiamato
- Il record di attivazione è messo sullo stack in modo tale che alla fine ritornino visibili le variabili del chiamante.
- Nel nuovo record, i parametri formali sono inizializzati con i valori ricavati al passo 2
- Per i metodi di istanza, la metavariabile **this** è inizializzata al riferimento ricavato al passo 1
- IMPORTANTE: dentro un metodo chiamato **non sono visibili le variabili del chiamante**

5 - Esecuzione del corpo

- Una volta identificato il corpo e creato il suo ambiente, l'esecuzione inizia dalla prima istruzione del corpo
 - prosegue poi come di consueto
 - termina alla fine del corpo o quando viene incontrato un comando **return**
 - può terminare anche in altri casi, in particolare se avviene un errore di qualche tipo (questo lo vedremo più avanti quando parleremo delle **eccezioni**)

6 - Distruzione del record di attivazione e ritorno

- Una volta terminata l'esecuzione del corpo, l'ambiente locale viene rimosso dallo stack
 - tutti i valori delle variabili locali vengono persi
 - è possibile che gli oggetti creati con **new** non siano più raggiungibili (divengano "garbage")
- Viene quindi ripristinato il record del chiamante
 - "ricompaiono" le variabili locali del chiamante, con i loro valori
- Il valore dell'espressione del **return**, se presente, diventa il valore della chiamata
 - ricordate che *InvocaMetodo* è una *ExprPri*
- L'esecuzione riprende quindi dal punto del chiamante in cui si trovava la chiamata

Metodi variadici

- È possibile definire metodi con un **numero variabile di parametri** (metodi **variadici**)

Params ::= *Param* (, *Param*)* | (*Param* (, *Param*)*)? *ParamVariad*

ParamVariad ::= *ModVarLoc*? *Tipo*...*Id*

- Questi metodi vengono poi invocati con 0 o più argomenti, tutti di tipo *Tipo*
- Il metodo chiamato vede gli argomenti variadici come un array di *Tipo*, cioè *Tipo*[]
- Esempio

- dichiarazione: **public int** max(**int** ... n)
- invocazioni: max(1,0,1000), max(), max(1)
- il body

```
public int max(int ... n) {  
    if (n.length > 0) {  
        int max=n[0];  
        for (int i:n) if (i>max) max=i;  
        return max;  
    } else return 0;  
}
```

Nota che **n** è considerato un array di **int**.

Note su metodi variadici

- il **parametro variadico** deve essere l'**ultimo** nella lista dei parametri formali
- può essere preceduto da 0 o più parametri non-variadici
- nella chiamata, al parametro formale variadico possono corrispondere 0 o più parametri attuali
- nel body del metodo deve essere considerato il caso di array di lunghezza 0
- Regola generale:
 - un metodo non-variadico n -ario deve essere chiamato con **esattamente n argomenti**
 - un metodo variadico n -ario deve essere chiamato con almeno $n - 1$ argomenti

Il metodo variadico printf

- Un esempio particolarmente importante di metodo variadico (usato per la stampa formattata) è

```
printf(String format, Object ... args)
```

- Il primo argomento è una **stringa di formato** (contiene dei segnaposto (indicati dal carattere %) che indicano dove stampare gli argomenti successivi)
- Gli argomenti variadici sono di tipo **Object** (in modo da poter fornire qualunque valore del linguaggio Java)
- Esempio

```
int z = 36;  
System.out.printf("z=%d (o %x in %s)", z, z, "hex");
```

produce

```
z=36 (o 24 in hex)
```

Esempio

Scriviamo un metodo `miaSomma` che calcola la somma dei suoi primi `k` parametri interi, dove `k` è il primo parametro del metodo.

La chiamata del metodo

`miaSomma(k, n1, , nm)`

deve ritornare

- $n_1 + \dots + n_k$ se $m > k$ (non si considerano i parametri dopo n_k)
- $n_1 + \dots + n_m$ se $m \leq k$

Metodi sovraccarichi (overloading)

- La **firma** di un metodo è il **nome seguito dalla sequenza dei tipi dei parametri** (il tipo del risultato non fa parte della firma!)
- Abbiamo visto che il meccanismo di **overloading** permette di usare lo stesso nome per più metodi.
 - che possono avere un corpo diverso
 - i metodi devono **differire nella firma** poichè il nome è lo stesso, deve essere diversa la sequenza dei tipi dei parametri
- La cosa importante dell'overlaoding è che il compilatore deve essere in grado di risolverlo, cioè capire quale metodo chiamare, a partire dai **tipi dei parametri**.

Regole di risoluzione

- Non sempre esiste una corrispondenza diretta fra firma della chiamata e firme dei metodi sovraccarichi
- Può essere necessario fare conversioni, promozioni, ecc.
- Se diverse conversioni sono possibili, si sceglie la più specifica (quella che richiede meno conversioni), ma occorre che sia unica altrimenti l'ambiguità causa un errore di compilazione

```
public long media(int x, long y) { ..... }  
public long media(long x, int y) { ..... }
```

Se chiamiamo `media(3,4)`, ci sono 2 possibili conversioni automatiche che possono essere applicate:

`media((long)3,4)`

`media(3,(long)4)`

per cui c'è ambiguità e il compilatore segnala errore.

Metodi generici

- Abbiamo visto finora come i metodi possano prendere dei **valori** come argomenti da associare ai **parametri**. Ad esempio
 - parametri formali: $T \ m(T_1 \ id_1, T_2 \ id_2)$
 - parametri attuali (argomenti): $m(v_1, v_2)$
- Java consente anche di definire metodi che prendano dei **tipi** come **parametri**. Ad esempio
 - parametri di tipo formali: $\langle id_a, id_b \rangle \ id_a \ m(T_1 \ id_1, id_b \ id_2)$
 id_a e id_b possono essere usati nel body di m come se fossero tipi normali
 - parametri di tipo attuali: $\langle T_a, T_b \rangle \ m(v_1, v_2)$
i tipi T_a e T_b devono essere **tipi riferimento**
- Si parla in questo caso di **metodi generici** in cui la genericità si riferisce ai tipi

Metodi generici: sintassi definizione

- Abbiamo visto che nella dichiarazione ci potevano essere parametri di tipo:

DichMetodo ::= Modificatore ParamsDiTipo? Tipo Id (Params) CorpoMetodo*

ParamsDiTipo ::= < ParamDiTipo (, ParamDiTipo) >*

ParamDiTipo ::= Id (extends TipoRif (& TipoRif))?*

- Nel caso più semplice, un *ParamDiTipo* è semplicemente un *Id*. Per convenzione, si usano singole lettere maiuscole.
- Un *ParamDiTipo* può anche assumere la forma più complessa che analizzeremo quando vedremo le classi generiche.

Esempio metodo generico

Scriviamo un metodo che fa il swap di due elementi di un array:

i parametri dovranno essere l'array e i due indici (interi) (in caso uno dei due indici sia fuori dall'array non si deve modificare l'array).

```
public static <T> void swap(T [] array, int i, int j)
```

Metodi generici: sintassi chiamata

- Abbiamo visto la sintassi del *RifMetodo* per metodi non generici. La sintassi che include anche i generici è come segue:

```
RifMetodo ::= Id | ( Id. )+ ArgDiTipo? Id | ExprPri . ArgsDiTipo? Id  
ArgsDiTipo ::= < ArgDiTipo ( , ArgDiTipo )* >  
ArgDiTipo ::= TipoRif
```

- prima del nome del metodo possono essere specificati gli argomenti di tipo che **devono essere tipi riferimento** racchiusi da parentesi angolate.
- Una chiamata del metodo definito nella pagina precedente può essere

```
Integer a1 []= {0,1,2,3};  
.....  
<Integer>swap(a1, 1, 3);
```

Java però permette anche la chiamata semplice

```
swap(a1, 1, 3);
```

perchè può inferire il tipo dal contesto!

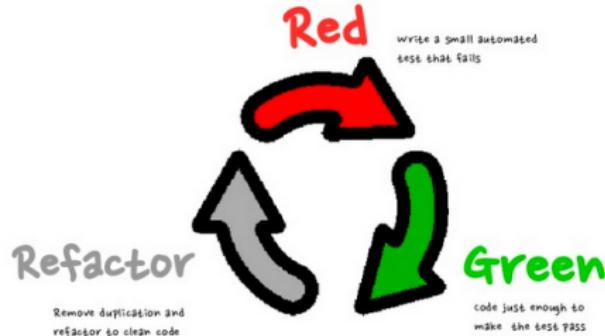
Corso: Paradigmi di Programmazione

Paola Giannini

Introduzione a Test Driven Development con Junit5
Esercizio 22 Ottobre

Che cosa è TDD?

- È una metodologia per sviluppare moduli di software nel nostro caso classi Java che consiste nello:
 - Scrivere **TEST** (la cui verifica è automatizzata) **PRIMA** del codice
 - Usare un Framework di testing (nel nostro caso Junit), che ci dice se i test sono verificati (**verde**) oppure no (**rosso**)
 - Modificare codice in piccoli passi, solo quando necessario, cioè quando **rosso**
- **Mantra:** **rosso**, **verde**, refactoring (ristruttura)



L'origine di TDD

- XP (**eXtreme Programming** - Kent Beck) è una metodologia usato nel sviluppo di software Agile (“Agile Software Development”)
- Una delle 12 “Best Practices” di XP è l’automazione di tutti i test
- L’ambito del TDD è quello del test di unità cioè testiamo i metodi delle classi (vedrete in SW che il testing è fatto a molti livelli)

Test

- I test vengono scritti prima del codice
- Sono rigorosamente **testati automaticamente**
- Vengono testati velocemente (secondi o minuti), per cui si possono testare molto frequentemente (diventa parte dello sviluppo normale)
- **Non si aggiunge mai funzionalità** a meno di non avere un test che **fallisce!**
- Migliora la progettazione
 - Piccoli test indipendenti implicano piccolo moduli indipendenti
 - Spinge a creare API, indipendenti dalla interfaccia utente
 - Procedimento che evita il “blocco dello scrittore”
 - Vengono prodotte allo stesso tempo due definizioni del programma: una descrittiva (cosa deve valere) e l'altra procedurale (come si fa)
 - I test servono come **documentazione** (per successivi sviluppatori) e come **definizione di requisiti**

Altre “Best Practices” di XP: DRY e “Shotgun Surgery”

- Il principio **DRY** (**Do not repeat yourself!**) viene spesso citato in relazione al **code smell** (letteralmente “puzza del codice”) associata alla **duplicazione del codice** ovvero il fatto che
 - codice **identico o simile esiste in più di una locazione del programma**
- Il “code smell” è una espressione usata per indicare una serie di caratteristiche che il codice sorgente può avere e che sono generalmente riconosciute come probabili indicazioni di un difetto di programmazione
- Un altro “code smell” è dato da quello che viene denominato **shotgun surgery**, cioè
 - un piccolo cambiamento di requisiti che viene realizzato **facendo cambiamenti in molte parti del codice**

Ulteriore documentazione

- eXtreme Programming : Kent Beck
<http://www.extremeprogramming.org/>
- Mock Objects
<http://www.mockobjects.com/>
- Refactoring:
 - Refactoring: Improving the Design of Existing Code, Fowler et al
 - Design Patterns: Elements of Reusable Object-Oriented Software, Gamma et al
 - Refactoring<https://www.refactoring.com/>

Esercizio: Gestione di una Rubrica

Si vuole realizzare una rubrica che contiene sequenze di stringhe del tipo:

"Giovanni=345778800", "Irene=016134599", ecc.

memorizzate in un vettore di stringhe, `ArrayList<String>` che ha una capienza massima (determinata da una costante `MAX_DIM`).

Le operazioni che vogliamo fare sulla rubrica sono:

- ① creare la rubrica (vuota) e testare che dopo la creazione la rubrica contenga 0 elementi
- ② svuotare la rubrica e testare che dopo averla svuotata la rubrica contenga 0 elementi
- ③ sapere quanti elementi ci sono
- ④ aggiungere una stringa se la rubrica non è piena e non contiene una stringa uguale (facciamo ritornare 1 se aggiungiamo, -1 se la rubrica è piena, 0 se non aggiungiamo perché già presente). Faremo un test per ogni situazione: se l'elemento viene inserito correttamente, se non viene inserito perché la rubrica è piena e se non viene inserito perché già presente nella rubrica.
- ⑤ ritornare TUTTE le stringhe che hanno come prefisso una data stringa
- ⑥ rimuovere TUTTE le stringhe che hanno un dato prefisso

Creiamo un nuovo progetto: PrimaRubrica, packages test e rubrica e la classe Rubrica

- NON ci sarà bisogno della classe Input per fare la lettura e NON definiremo un programma (classe con **static void** main(...))
- Vogliamo tenere i codice organizzato per cui definiamo **2 packages**:
 - rubrica (che conterrà la classe Rubrica)
 - test (che conterrà la classe di test)
- Nel package rubrica create solamente la classe Rubrica e inseriamoci solo il contenitore per la rubrica e la dimensione massima.

```
public class Rubrica {  
    private static ArrayList<String> rubrica;  
    public static final int MAX_DIM = 3;  
}
```

Iniziamo definendo una classe di test, RubricaTest.

```
New > JUnit Test Case  
> OK per Add Add JUnit library
```

Accertatevi che

- La classe RubricaTest sia nel package test

Dovreste ottenere

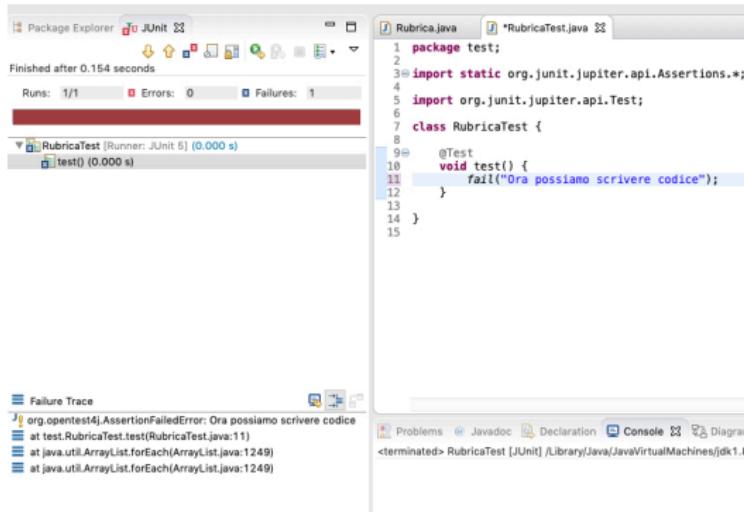
```
package test;  
  
import static org.junit.jupiter.api.Assertions.*;  
import org.junit.jupiter.api.Test;  
  
class RubricaTest {  
  
    @Test  
    void test() {  
        fail("Not yet implemented");  
    }  
}
```

potete modificare la stringa "Not yet implemented"! Ad esempio rimpiazziamo con
"Ora possiamo scrivere codice"

Abbiamo già scritto un test e possiamo eseguirlo

- Potete eseguire il test (cioè il metodo `test`) con il solito bottone (triangolo bianco nel bottone verde) oppure cliccando il tasto destro posizionato sul metodo e selezionando:

Run As > Junit Test



Cosa succede se commentiamo l'istruzione fail("...")?

- Commentiamo `fail("Ora possiamo scrivere codice");` e eseguiamo il test otteniamo un successo. Perchè?

The screenshot shows the Eclipse IDE interface with the JUnit perspective active. The top bar displays "Package Explorer" and "JUnit". The status bar indicates "Finished after 0.125 seconds", "Runs: 1/1", "Errors: 0", and "Failures: 0". The code editor shows two files: Rubrica.java and RubricaTest.java. Rubrica.java contains a simple class definition. RubricaTest.java contains a test method that is currently commented out with a block comment block:

```
1 package test;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 import org.junit.jupiter.api.Test;
6
7 class RubricaTest {
8
9     @Test
10    void test() {
11        // fail("Ora possiamo scrivere codice");
12    }
13
14 }
```

The "Failure Trace" view is open at the bottom left, showing no errors or failures. The bottom navigation bar includes "Problems", "Javadoc", "Declaration", "Console", and "Diagrams". The "Console" tab shows the message "<terminated> RubricaTest (1) [JUnit] /Library/Java/JavaVirtualMachines/jdk1".

Cominciamo a scrivere i test

- Iniziamo dal test di creazione:

```
@Test  
void testCreazione() {  
    Rubrica.crea();  
}
```

- Facciamo generare dal framework il metodo statico `crea`. E nel metodo `crea` inizializziamo la variabile `rubrica` a una `new ArrayList<String>()`.
- Giriamo i test di creazione:

VERDE

- In effetti **NON abbiamo testato niente!** Però abbiamo fatto eseguire il metodo `Rubrica.crea()` e sappiamo che non genera errori!

Le asserzioni: metodi assert.....

- Abbiamo detto che dopo la creazione il numero degli elementi della rubrica deve essere uguale a 0. Quindi scriviamo la nostra prima asserzione:

```
@Test  
void testCreazione() {  
    Rubrica.crea();  
    assertTrue(Rubrica.numEl()==0);  
}
```

- Facciamo generare dal framework il metodo statico numEl() e implementiamo il metodo nel modo ovvio!
- Il test è VERDE
- Avremo potuto usare

```
assertEquals(int expected, int actual)
```

cioè

```
assertEquals(0, Rubrica.numEl())
```

Note su metodi assert..... e loro esecuzione

- Ci sono molti metodi assert:

<https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html>

- I metodi assert possono avere anche un parametro aggiuntivo di tipo String nel quale potete mettere un messaggio che viene stampato se l'asserzione non vale! Es.

```
assertTrue(Rubrica.numEl()==0, "Rubrica.numEl() != 0");
```

- Se un metodo assert fallisce le istruzioni seguenti (e quindi eventuali metodi assert che seguono) NON vengono eseguiti

Scriviamo un test per il metodo che svuota la rubrica

```
@Test  
void testSvuota() {  
    Rubrica.svuota();  
    assertTrue(Rubrica.numEl()==0);  
}
```

- Facciamo generare dal framework il metodo statico svuota().

Annotazioni per test e ciclo di esecuzione

- Ci sono 4 annotazioni (principali) per metodi coinvolti nei test e che ne regolano l'esecuzione: `@BeforeEach`, `@AfterEach`, `@BeforeAll` e `@AfterAll`.
- I metodi di test sono annotati con `@Test` e sono quelli che contengono le asserzioni (le chiamate di metodi `assert....`)
- Ogni metodo di test è **eseguito indipendentemente dall'altro**. Non c'è un ordine specificato!
 - ① Prima di ogni metodo annotato con `@Test` vengono eseguiti, se ci sono i metodi annotati con `@BeforeEach`
 - ② Dopo ogni metodo annotato con `@Test` vengono eseguiti, se ci sono i metodi annotati con `@AfterEach`
- `@BeforeAll` e `@AfterAll` (che devono essere statici) vengono eseguiti 1 sola volta prima di iniziare l'esecuzione di tutti i metodi e dopo l'esecuzione di tutti i metodi (rispettivamente)

Ristrutturiamo la classe di test

```
class RubricaTest {  
  
    @BeforeEach  
    void setup() {  
        Rubrica.crea();  
    }  
  
    @AfterEach  
    void reset() {  
        Rubrica.svuota();  
    }  
  
    @Test  
    void testCreazione() {  
        assertTrue(Rubrica.numEls() == 0);  
    }  
  
    @Test  
    void testSvuota() {  
        Rubrica.svuota();  
        assertEquals(0, Rubrica.numEls());  
    }  
}
```

Ritorniamo a testare la classe: il metodo aggiungi

Scrivete 3 metodi di test per aggiungi:

- ① uno che aggiunge correttamente (e che testa che dopo l'aggiunta ci sia il numero di elementi giusto)
 - ② uno che aggiunge un elemento in più di MAX_DIM e che quindi non aggiunge l'elemento (ritorna -1) e fa vedere che anche dopo l'aggiunta ci sono MAX_DIM elementi (questo test vi autorizza ad aggiungere codice)
 - ③ uno che aggiunge un elemento che c'è già e che non lo deve aggiungere (ritornare 0) e non modificare la dimensione (questo test vi autorizza ad aggiungere codice).
-
- ① Iniziate dal test per l'esecuzione corretta e scrivete il minimo codice indispensabile per farlo passare (cioè aggiungete a rubrica senza fare controlli). Testate il valore di ritorno e il numero degli elementi!
 - ② Poi scrivete il test che controlla che non vengano aggiunti più elementi di MAX_DIM, facendolo fallire! Modificate il codice di aggiungi
 - ③ Infine fate l'ultimo

Ora che avete capito come si fa

Passiamo al metodo `ricerca`. Deve ritornare una lista `ArrayList<String>`.
Come fare i test

- ① Inserire un po' di elementi
- ② Poi cercare e testare che venga ritornato il numero giusto di elementi
- ③ Provate la ricerca nella rubrica vuota/ piena
- ④ Mettendo un elemento in prima/ultima posizione

IMPORTANTE: I metodi di test devono essere INDIPENDENTI perchè non è assicurato un ordine di VALUTAZIONE. Si sa solamente che `@BeforeEach` è eseguito prima di ogni metodo `@Test`

L'ultimo metodo della rubrica

Scrivere il metodo di eliminazione dalla Rubrica. Il metodo elimina ha il seguente prototipo

```
public static boolean elimina(String s)
```

e deve rimuovere dalla rubrica tutte le stringhe che iniziano per s e ritornare **true** se elimina almeno una stringa e **false** altrimenti

Pensare ai test da fare!

Corso: Paradigmi di Programmazione

Paola Giannini

La programmazione orientata agli oggetti

- Visione **tradizionale**

- La computazione è l'esecuzione in un certo ordine di operazioni sui dati. L'ordine è dato dai costrutti di controllo
- Le strutture dati sono parametri delle funzioni (computazioni)

- Visione **object-oriented**

- La prospettiva è rovesciata
- Gli oggetti (i dati!) sono gli attori principali della computazione
- La computazione è data dallo scambio di messaggi fra gli oggetti

La filosofia OO (1)

- Le classi corrispondono nella visione OO a categorie di oggetti che esistono nel dominio del problema
- Sono buoni candidati
 - Persone, Studenti, Professori, Corsi, Aule, ...
 - Banche, Clienti, Conti correnti, Mutui, ...
 - Automobili, Caselli, Tariffe, Telepass, ...
- Gli oggetti contengono
 - Uno **stato** (le variabili che gestiscono)
 - I **metodi che manipolano lo stato**
- Ogni operazione che implica l'uso dello stato di un oggetto dovrebbe essere eseguita da un metodo dell'oggetto!

La filosofia OO (2)

- La computazione procede attraverso l'invio di messaggi
 - un oggetto invia un messaggio a un altro oggetto per richiedere a quest'ultimo una qualche operazione (sul suo stato)
 - un messaggio ha un nome e un insieme di parametri
 - l'oggetto destinatario esegue una parte del proprio codice (un metodo) in risposta al messaggio
 - il risultato viene restituito (come risposta) all'oggetto mittente
- Il programmatore non ragiona sul flusso complessivo dell'esecuzione.
Piuttosto, si concentra su
 - quali oggetti debbano esistere nel sistema
 - quali operazioni ogni oggetto debba implementare
 - quali messaggi debbano essere riconosciuti
 - quali operazioni di altri oggetti possano essere utili a un oggetto nell'implementare le proprie operazioni

I quattro strumenti dell'OO

- Indipendentemente dal linguaggio OO utilizzato, possiamo individuare quattro “strumenti concettuali” per una efficace strutturazione del codice OO
 - Ereditarietà
 - Astrazione
 - Incapsulamento
 - Polimorfismo

Ereditarietà

- le classi sono organizzate in **gerarchie**
- le **sottoclassi** ereditano tutte le caratteristiche delle loro **superclassi**
 - possono poi modificarle o aggiungere caratteristiche proprie
 - il codice che implementa le operazioni di una classe è a disposizione di tutte le sue sottoclassi
- Esempio
 - Veicolo è una superclasse per Automobile e Ciclomotore
 - Ogni Automobile (o Ciclomotore) è un Veicolo. Questa relazione è a volte indicata come **is-a** (**è-un**)

- solo informazioni rilevanti
 - consiste nel limitare i dettagli usati per modellare un oggetto a quelli effettivamente necessari per soddisfare i requisiti. Se consideriamo un'applicazione bancaria oppure una medica e supponiamo di dover modellare gli utenti
 - nel primo caso altezza e peso non saranno rilevanti
 - nel secondo caso lo saranno
- uso del livello corretto nella gerarchia di ereditarietà
 - se devo esprimere qualcosa valido solo per Automobile, uso l'oggetto Automobile - ma se si tratta di informazione più generale, uso l'oggetto Veicolo
- uso della composizione
 - un oggetto può fungere da contenitore; può offrire operazioni all'esterno e implementarle delegandone l'esecuzione alle sue parti. Quando la nostra Rubrica conterrà dei Contatti un po' più strutturati delle semplici stringhe delegherà a questi oggetti i controlli che vogliamo fare (**separazione di responsabilità**).

- protezione dello stato
 - rendo visibile/accessibile all'esterno lo stato SOLO attraverso
- protezione delle funzionalità
 - rendo visibili/accessibili all'esterno solo alcune delle operazioni che l'oggetto offre; tutte le altre vengono nascoste al suo interno
- L'incapsulamento consente di fornire all'esterno una interfaccia pubblica ben specificata, che non faccia trasparire i dettagli dell'implementazione, e sia di facile uso per il chiamante

Polimorfismo

- polimorfismo per **ereditarietà**
 - un'istanza di una classe può essere trattata come se fosse un'istanza di una qualunque delle sue superclassi (l'operazione eseguita dipende dalla classe "vera" dell'oggetto, non dalla classe usata per riferirlo)
- polimorfismo **parametrico**
 - una classe può essere generica o definire metodi generici (l'operazione eseguita dipende dai parametri di tipo forniti)

I quattro strumenti

- I principi per l'applicazione di questi strumenti nella modellazione e progettazione di sistemi sono l'oggetto di studio dell'**Ingegneria del Software**
- I quattro strumenti sono validi **indipendentemente dal linguaggio** di programmazione utilizzato
 - è possibile, per esempio, realizzare un sistema object-oriented che esibisca l'uso dei quattro strumenti anche in linguaggi che non sono object-oriented, come il C
- Vedremo ora come i quattro strumenti sono realizzati nel linguaggio Java (argomenti che approfondiamo nel corso)

La programmazione OO in Java

Concetto OOP	Costrutto Java
Classi	Classi e .class
Oggetti	Oggetti, operatore new, operatore .
Messaggi	Metodi
Ereditarietà	Ereditarietà singola, clausola extends
Astrazione	Scelta variabili (statiche, istanza) Uso corretto della gerarchia. Uso di interfacce. Dichiarazione di un oggetto come componente di un altro
Incapsulamento	Modificatori di visibilità public, protected, private; package e import; interface e implements
Polimorfismo	upcast e downcast; overriding;overloading

- Abbiamo tre entità:
 - classi, oggetti, messaggi
- quattro strumenti
 - ereditarietà, astrazione, encapsulamento, polimorfismo
- Per ciascuno di essi, Java offre dei costrutti linguistici

OO in Java: messaggi

- I messaggi sono realizzati in Java tramite l'**invocazione di metodo**
- Nome e parametri del metodo = nome e parametri del messaggio
- Valore di ritorno (dal return) = risultato dell'operazione
- Si tratta di messaggi **sincroni** e **bloccanti**
 - l'esecuzione del mittente/chiamante viene sospesa finchè non è terminata l'elaborazione del messaggio/metodo da parte del destinatario/chiamato

Principali linguaggi OO

- **Simula**: progettato nel 1967 - capostipite dei linguaggi ad oggetti (nasce dal modo della simulazione)
- **Smalltalk**: anni 80 - linguaggio ad oggetti "puro" (nasce come linguaggio delle prime workstation con interfaccia grafica)
- **C++**: linguaggio "ibrido" ? estende il C (con il quale è compatibile) con costrutti OO
- **Java**: creato dalla Sun come linguaggio per la programmazione della rete (sicurezza e portabilità)
- **C#**: simile a Java, alla base della tecnologia **.Net** di Microsoft.
- **JavaScript**: linguaggio di scripting che è immerso in pagine HTML
- **VisualBasic**: linguaggio di scripting per applicazioni Windows.
- **Python**: linguaggio **multiparadigma** usato sia per lo scripting, ma con ottimo supporto per la gestione di stringhe, ecc...
- **Scala**: linguaggio **multiparadigma** usato per il processamento di Big Data
- **Ruby**: usato per la produzione di siti RubyOnRails

Come si programmano gli oggetti

- Due tipi di linguaggi ad oggetti:

- Quelli **basati su classi**. (quasi tutti i precedenti eccetto JavaScript e VisualBasic). Si programmano classi che definiscono il comportamento di insiemi di oggetti. Una operazione fondamentale è la **creazione** di un oggetto a partire da una classe.
- Quelli **basati su oggetti**. Si programmano direttamente oggetti. In genere si parte da **prototipi** (oggetti di sistema o semplici oggetti definiti da utente) e si usano e/o aggiungono e/o modificano metodi e campi su questi oggetti.

Tipi e Oggetti

- Nei linguaggi basati su classi, le **classi sono i tipi** del linguaggio (in C# anche i tipi primitivi sono oggetti istanza di classi)
- I linguaggi basati su oggetti in generale non hanno tipi a tempo di compilazione. Si parla di **“duck typing”**:

se cammina come un'anatra, nuota come un'anatra e starnazza come un'anatra, allora è un'anatra

Corso: Paradigmi di Programmazione

Paola Giannini

Sviluppo classi

Dichiarazione di classi

- La dichiarazione di una classe, nella forma più generale, ha la seguente struttura:

DichClasse ::= Modificatore **class** Id ParamsDiTipo?*
Estende? Implementa? CorpoClasse

- *Estende* permette di indicare la classe da cui si eredita
- *Implementa* permette di indicare le interfacce che si implementano
- *Estende* e *Implementa* saranno analizzate in seguito.
- *ParamsDiTipo* è una sequenza di **variabili di tipo**, eventualmente con vincoli, se presenti la classe è **generica** abbiamo già usato classi generiche, es. `ArrayList<T>` e vedremo come dichiarale più avanti.

I modificatori di classe

I *Modifieri* ammissibili nell'intestazione di una classe sono:

- **public**: la classe è visibile anche **fuori dal package** dove è dichiarata
- “nessun modificatore” è visibile **solo nel package** in cui è dichiarata
- **abstract, final**: legati all'**ereditarietà**
- **protected, private, static** ammissibili solo per **classi membro** (cioè classi dichiarate all'interno di altre classi)

Il corpo di una classe

- Il corpo di una classe contiene una sequenza di dichiarazioni che introducono variabili di classe e d'istanza , costruttori , metodi di classe e d'istanza , classi o interfacce membro , tipi enumerativi e blocchi di inizializzazione

CorpoClasse ::= { DichInClasse }*

*DichInClasse ::= DichCampo | DichCostruttore | DichMetodo | DichClasse
| DichInterfaccia | DichEnum | DichIniClasse | DichIniStanza*

- Nel seguito, statico e di classe sono sinonimi

Contesti statici e non statici (1)

- Rivediamo la classe PuntoColorato:

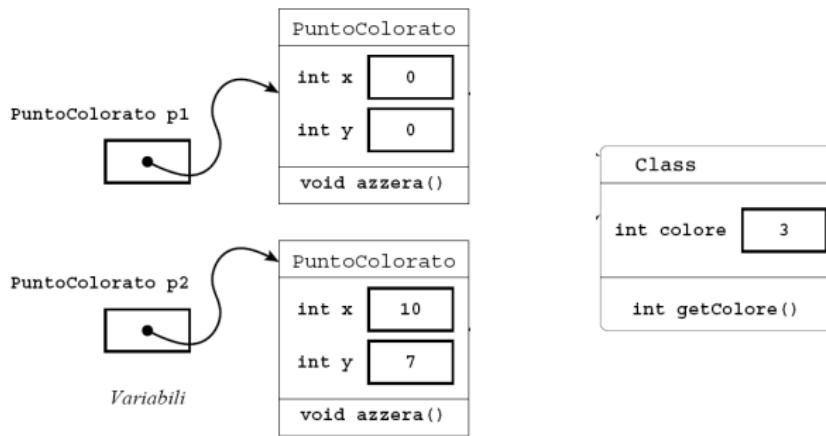
```
1. public class PuntoColorato {  
2.     public static int colore;  
3.     public int x,y;  
4.     public void azzera() {  
5.         x=0; y=0; colore=0;  
6.     }  
7.     public static int getColore() {  
8.         return colore;  
9.     }  
10.}
```

- Per ora abbiamo visto che le **istruzioni/espressioni** possono comparire **nei metodi** oppure nelle **inizializzazioni dei campi**.
- A tempo di esecuzione, un'istruzione/espressione che compare in una classe sarà eseguita
 - in un **contesto statico** (cioè disponendo solo della “metaclasse”), oppure
 - in un **contesto non statico** (cioè “in un oggetto della classe”, chiamato l'**istanza corrente**)
- Le istruzioni nella riga 5. sono eseguite in contesto non statico, mentre l'espressione nella riga 8. in contesto statico

Contesti statici e non statici: esempio

Considera il codice

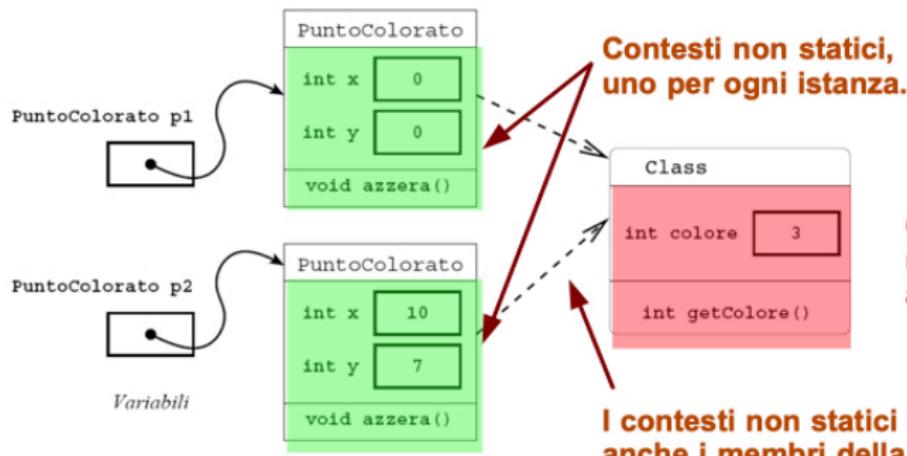
1. PuntoColorato p1 = **new** PuntoColorato();
2. PuntoColorato p2 = **new** PuntoColorato();
3. p2.x = 10;
4. p2.y = 9;
5. PuntoColorato.colore = 3;



Contesti statici e non statici: esempio

Considera il codice

1. PuntoColorato p1 = **new** PuntoColorato();
2. PuntoColorato p2 = **new** PuntoColorato();
3. p2.x = 10;
4. p2.y = 9;
5. PuntoColorato.colore = 3;



Variabili d'istanza e di classe

- Una classe può dichiarare variabili **di classe** (o **statiche**) e variabili **d'istanza** (chiamati anche **campi**)
- La sintassi è simile alla dichiarazione di variabili locali di un metodo:

DichCampo ::= Modificatore Tipo Id DichVar⁺*
DichVar ::= Id (= EsprIniVar?)?

- Una variabile è **di classe** se *Tipo* è preceduto dal modificatore **static**, altrimenti è **d'istanza**
- EsprIniVar* è un'**espressione di inizializzazione** (come per le variabili locali). Se manca è come se fosse specificato il default per il tipo
 - 0 se tipo primitivo numerico
 - false** se tipo **boolean**
 - null** in tutti gli altri casi (tipi riferimento e di conseguenza anche per i wrapper dei tipi primitivi!)
- A tempo di esecuzione esiste
 - un'unica copia di ogni variabile statica
 - per ogni oggetto, una copia di ogni variabile d'istanza

Modificatori ammissibili e significato

- I seguenti modificatori (oltre a **static**) possono comparire in una dichiarazione di campo:
 - final** indica, come per le variabili locali, che si tratta di una **costante**, assegnabile una sola volta
 - public, protected, private** e **nessun modificatore** determinano il livello di accessibilità (pubblico, protetto, privato e default)
 - transient** e **volatile** sono legati a “serializzazione” (scrittura degli oggetti su memoria esterna alla JVM) e “multithreading”

Dichiarazione di campi: esempio

```
public enum Colore {NERO,ROSSO,VERDE,GIALLO};

1. public class PuntoColorato {
2.     public static final Colore defColor = Colore.NERO;
3.     public static Colore colore;
4.     private int x, y;
5.     public final int defVal = 0;
6.     public void setDefault() {
7.         x=defVal; y=defVal; colore=defColor;
8.     }
9.     public int getX() {
10.         return x;
11.     }
12.     public int getY() {
13.         return y;
14.     }
15.     public void setX(int x) {
16.         if (x>=0) this.x = x;
17.     }
18.     public void setY(int y) {
19.         if (y>=0) this.y = y;
20.     }
21. }
```

Accesso a campi

- Con **dot-notation**

- *NomeClasse.NomeCampo* per una variabile statica:

PuntoColorato.colore

(o *Math.PI* è la costante statica della classe *Math* che contiene un valore approssimato di pi-greco)

- *Oggetto.NomeCampo* per una variabile d'istanza

```
PuntoColorato p = new PuntoColorato();  
int d= p.defVal;
```

- Con **nome semplice**. Si può accedere a un campo di una classe con il suo nome semplice:

- solo all'interno del suo ambito di visibilità, che comprende tutta la classe in cui è dichiarato
- si può accedere a una variabile d'istanza (dell'istanza corrente) solo da un **contesto non statico**
- Se il nome del campo è nascosto dal nome di un parametro formale o da una variabile locale si può usare:
NomeClasse.NomeCampo per una variabile statica
this.*NomeCampo* per una variabile d'istanza dell'istanza corrente

Accesso a campi: esempio

```
public enum Colore {NERO,ROSSO,VERDE,GIALLO};

1. public class PuntoColorato {
2.     public static final Colore defColor = Colore.NERO;
3.     public static Colore colore;
4.     private int x, y;
5.     public final int defVal = 0;
6.     public void setDefault() {
7.         x=defVal; y=defVal; colore=defColor;
8.     }
9.     public int getX() {
10.         return x; //accesso senza dot-notation
11.     }
12.     public int getY() {
13.         return y; //accesso senza dot-notation
14.     }
15.     public void setX(int x) {
16.         if (x>=0) this.x = x; //accesso tramite this perche' oscurato da parametro
17.     }
18.     public void setY(int y) {
19.         if (y>=0) this.y = y; //accesso tramite this perche' oscurato da parametro
20.     }
21. }
```

I costruttori

- Quando si crea un oggetto, la **new** invoca un **costruttore** su di esso
- I costruttori tipicamente **completano l'inizializzazione** di un oggetto, es:
inizializzando le variabili d'istanza, controllando la consistenza...
- Hanno una sintassi simile ai metodi, ma:
 - Il nome *Id* deve essere lo stesso della classe
 - Non va indicato un *Tipo* del risultato

DichCostruttore ::= *Modificatore** *Id* (*Params*?) *CorpoCostruttore*
CorpoCostruttore ::= *InvocaCostruttore*? *CmdInBlocco**

- I Modificatori ammissibili sono solo **public**, **protected** o **private** oppure **nessun modifikatore**
- Il corpo del costruttore viene eseguito in un contesto non statico, dove l'istanza corrente è l'oggetto che verrà restituito dalla **new**
- *InvocaCostruttore* serve per invocare un altro costruttore, come vedremo. Se c'è viene eseguito in contesto statico

I costruttori: esempio

```
public PuntoColorato(int x,int y, Colore colore) {  
    setX(x);  
    setY(y);  
    PuntoColorato.colore=colore;  
}
```

- Questo costruttore della classe PuntoColorato inizializza le variabili d'istanza x e y e quella d'istanza colore con i valori dei parametri attuali.
- Poichè il parametro colore ha nome uguale al campo si scrive PuntoColorato.*NomeCampo* per il campo statico.
- Usando i metodi set per inizializzare le variabili di istanza se uno dei parametri è minore di zero la corrispondente variabile di istanza viene inizializzata a zero.

Creazione di istanze

- Sintassi di un'espressione **new**, già vista precedentemente, è:

<i>EsprCrealstanza</i>	$::=$	new	<i>IdQ</i>	(<i>Args?</i>)	parametri attuali
<i>Args</i>	$::=$	<i>Espressione</i>	(, <i>Espressione</i>)*	nome qualificato della classe	

- ## ● Esempio

```
int x=2;
PuntoColorato p1 =
    new PuntoColorato((2*3),(x+3),Colore.GIALLO);
```

Overloading e costruttore default

- Per l'**overloading**, una classe può avere un numero arbitrario di costruttori purchè abbiano firme diverse
- Il costruttore da eseguire è determinato, come per i metodi (che vedremo fra poco), dalla firma dell'invocazione, cioè da tipo e numero dei parametri attuali nell'espressione **new**
- Ogni classe ha almeno un costruttore. Se esso manca, Java fornisce il **costruttore default**, senza parametri, che non fa niente (ricordate l'inizializzazione dei campi ai valori di default è fatta dopo aver allocato la memoria!)
- Se si specifica un costruttore allora Java NON ci fornisce più il costruttore con 0 argomenti. Ad esempio, per la classe PuntoColorato a pag.12 possiamo costruire un oggetto con il costruttore di default:

```
new PuntoColorato()
```

se però aggiungiamo il costruttore di pag. 14 potremo solo generare oggetti

```
new PuntoColorato(3,2,Colore.NERO)
```

Invocazione di costruttore con `this(...)`

- La prima istruzione di un costruttore può essere un'invocazione esplicita di costruttore, con la primitiva `this(.....)`

*CorpoCostruttore ::= InvocaCostruttore? CmdInBlocco**
InvocaCostruttore ::= this (Args?);

- Esempio

```
public PuntoColorato() {  
    this(0,0,defColor);  
}
```

- `this(_)` deve essere il primo comando del costruttore
- viene eseguito in un **contesto statico**, perciò

```
public PuntoColorato() {  
    this(defVal,defVal,defColor); // ERRORE !!!  
}
```

il suggerimento (sensato) del compilatore è “**Change `defVal` to `static`**”.
Perchè?

I blocchi di inizializzazione

- In una classe possono comparire uno o più blocchi di inizializzazione, cioè blocchi di comandi al top level, eventualmente preceduti da **static**

DichInIstanza ::= Blocco

DichInIClasse ::= static Blocco

- I blocchi di inizializzazione statici vengono eseguiti nell'ordine (in un **contesto statico**) quando viene **caricata la classe**
- I blocchi di inizializzazione d'istanza vengono eseguiti nell'ordine quando viene **costruita un'istanza** della classe, in un **contesto non statico**, prima dell'esecuzione **del costruttore**

Esempio Blocchi di inizializzazione

```
class Test {  
    static int i;  
    int j;  
  
    static { i = 10; System.out.println("static block called i= " + i);}  
  
    { j = 20; System.out.println("non static block called i= " + i + ", j= " + j); i++; j++}  
  
    Test() {System.out.println("Constructor called");}  
}  
  
public class Main {  
    public static void main(String args[]) {  
        // Lo static block e' eseguito una volta mentre il non static due volte.  
        Test t1 = new Test();  
        Test t2 = new Test();  
    }  
}
```

Il risultato dell'esecuzione produce:

```
static block called i= 10  
non static block called i= 10, j= 20  
Constructor called  
non static block called i= 11, j= 20  
Constructor called
```

I metodi

- I metodi dichiarati in una classe descrivono funzionalità della classe (se statici) o dei suoi oggetti (se d'istanza).
- Abbiamo già visto in dettaglio come si dichiarano ed eseguono i metodi.
- Vi ricordo che:
 - Un metodo statico può essere invocato dall'**interno della classe** con `nomeMetodo(...)`, oppure dall'**esterno** con `NomeClasse.nomeMetodo(...)`, se accessibile
 - Un metodo d'istanza può essere invocato sull'**istanza corrente di un contesto non statico** con `nomeMetodo(...)`, o su un oggetto `obj` della classe con `obj.nomeMetodo(...)`

I modificatori di metodi

- I Modificatori ammissibili, oltre **static**, sono
 - quelli di visibilità (**public**, **protected**, **private** e nessun modifikatore)
 - **abstract** e **final** (diverso significato da **final** per un campo), li vedremo nel seguito
 - **synchronized**: permette mutua esclusione in caso di **multithreading**. La JVM garantisce che non più di un thread alla volta esegua un metodo **synchronized** su uno stesso oggetto (lo vedremo più avanti)
 - **native**: indica che il metodo è implementato da codice nativo **dipendente dalla piattaforma**; in questo caso il corpo deve essere ;

Le tecniche di incapsulamento

- L'**incapsulamento** consiste nel definire una precisa interfaccia tramite cui entità esterne possono interagire con una classe e le sue istanze
- Si possono ottenere vari livelli di incapsulamento con uso di package e di modificatori di accessibilità
- Si potrebbe volere, per esempio:
 - limitare l'accesso a una classe;
 - quelli di visibilità (**public**, **protected**, **private** e nessun modificatore)
 - disciplinare l'accesso a variabili statiche o d'istanza;
 - consentire l'invocazione di certi metodi solo in determinati contesti;
 - limitare la creazione di nuove istanze.

I modificatori di visibilità per le definizioni di classi

- I modificatori determinano 4 livelli di accessibilità:
 - ① **public**: pubblico
 - ② nessun modificatore: default
 - ③ **private**: privato
 - ④ **protected**: protetto
- Quando si riferiscono alla dichiarazione di una **classe**
 - ① Una classe **pubblica** è visibile dappertutto. Può essere riferita dall'esterno del package con il suo nome qualificato `nomePackage.NomeClasse` e con il nome semplice `NomeClass` dall'interno del suo package.
 - ② Una classe con accessibilità **default** è visibile solo all'interno del proprio package
 - ③ Una classe con accessibilità **privata** è visibile solo all'interno della classe in cui è definita
 - ④ Una classe con accessibilità **protetta** è visibile all'interno della classe in cui è definita e nelle sue sottoclassi

I modificatori di visibilità per le definizioni di membri

- Sia m un **membro** o **componente** (**costruttore**, **metodo** o **variabile**) di una classe C , dichiarata nel package p
- Valgono le seguenti regole:
 - All'interno della classe C , m è sempre visibile
 - In una classe del package p diversa da C , m non è visibile solo se è dichiarata **private**
 - In una classe che eredita da C ed è esterna al package p , m è visibile solo se C è **public** ed m è **public** o **protected**
 - In una classe esterna al package p e che non eredita da C , m è visibile solo se sia C che m sono **public**

Regole di accessibilità per membri di classe: esempio

```
package p1;

public class A {
    public int c1;
    private int c2;
    protected int c3;
    int c4;
    .....
}

class B extends A {
    void m(){
        c1 =1;
        c2 =1; //ERRORE
        c3 =1;
        c4 =1;
    }
    .....
}

class C {
    void m(){
        A a;
        a.c1 =1;
        a.c2 =1; //ERRORE
        a.c3 =1;
        a.c4 =1;
    }
    .....
}
```

```
package p2;

class D {
    void m(){
        A a;
        a.c1 =1;
        a.c2 =1; //ERRORE
        a.c3 =1; //ERRORE
        a.c4 =1; //ERRORE
    }
    // .....
}

class E extends p1.A {
    void m(){
        A a;
        a.c1 =1;
        a.c2 =1; //ERRORE
        a.c3 =1;
        a.c4 =1; //ERRORE
    }
    // .....
}
```

Limitare l'accesso alle variabili

- Spesso si vuole garantire che i valori che una variabile può assumere soddisfino certi vincoli.
- Nel caso della classe PuntoColorato, ad esempio
 - vogliamo che le coordinate dei punti non siano negative
- Una buona prassi consiste nel dichiarare tutte le variabili (statiche o di istanza) **private**, e nel fornire opportuni **metodi accessori** per leggerne e/o modificarne il valore. Questi metodi dovranno garantire che i vincoli siano soddisfatti.

Uso di metodi accessori

- Filtrando l'accesso a una variabile attraverso i metodi accessori, possiamo facilmente imporre delle politiche di accesso: come nel nostro esempio

```
public void setX(int x) {  
    if (x>=0) this.x = x;  
}  
public void setY(int y) {  
    if (y>=0) this.y = y;  
}
```

- Oppure se volessimo avere **punti immutabili** (come gli oggetti di tipo String) potremmo
 - ① eliminare i metodi setX e setY,
 - ② oppure definire setX e setY in modo tale da generare un nuovo PuntoColorato come segue

```
public PuntoColorato setX(int x) {  
    if (x>=0) return new PuntoColorato(x,this.y);  
}  
public PuntoColorato setY(int y) {  
    if (y>=0) return new PuntoColorato(this.x,y);  
}
```

Nota che questo è quello che succede per i metodi della classe String. Ad esempio per **replace**.

Corso: Paradigmi di Programmazione

Paola Giannini

Ereditarietà e classi astratte

Ereditarietà: Sottoclassi e superclassi

- L'**ereditarietà** permette di sviluppare una nuova classe in modo incrementale, riusando e/o specializzando funzionalità esistenti e aggiungendone di nuove
- Per dichiarare che una classe **eredita da** (o **estende**) un'altra classe, si usa la clausola *Estende* nell'intestazione

DichClasse ::= *Modificatore** **class** *Id ParamsDiTipo*?
Estende? **Implements**? *CorpoClasse*

Estende ::= **extends** *IdQ ArgsDiTipo*?

- Esempi

```
public class subC extends C
{
    /* corpo della sottoclasse */
}
public class IntList extends java.util.ArrayList<Integer>
{
    /* corpo della sottoclasse */
}
```

Significato di `extends`

```
public class subC extends C {  
    /* corpo della sottoclasse */  
}
```

- SubC è una **sottoclasse** o **classe derivata** da C
- C è la superclasse di SubC
- SubC **eredita** tutti i membri di C e delle sue superclassi: dichiarazioni di variabili, di metodi e di classi membro, sia statici che di istanza
- SubC **non eredita** blocchi di inizializzazione e costruttori
- SubC **estende** C con i suoi nuovi membri

Ereditarietà come inclusione

- Se una classe rappresenta l'**insieme** di tutte le sue istanze, **extends** rappresenta l'**inclusione** tra insiemi
 - quindi ogni istanza di SubC è anche un oggetto di C
- Ne segue la seguente regola fondamentale di Java, **principio di sostituzione di Liskov**¹:
Se SubC è una sottoclasse di C, allora in ogni contesto in cui è richiesta un'istanza di C si può utilizzare un'istanza di SubC
- Per esempio, il seguente assegnamento è valido:
`C var = new SubC()`
- Quindi il tipo di una sottoclasse è compatibile per assegnamento verso il tipo della superclasse

¹Barbara Liskov, 2008 Turing Award

La superclasse di tutte le classi: Object

- Ogni classe può essere estesa (da un numero arbitrario di classi), purché non sia dichiarata **final**
 - Es: si provi a dichiarare una classe che estende `String`
- Se nell'intestazione di una classe manca la clausola **extends**, Java la considera come sottoclasse diretta di `java.lang.Object`, che è quindi la radice della gerarchia di ereditarietà
- `Object` contiene metodi di utilità generale, disponibili in tutte le classi di Java

Ereditarietà: un esempio

- Una classe Punto ha variabili di istanza **private** x e y, metodi accessori, e un metodo `getDistanzaOrig()`

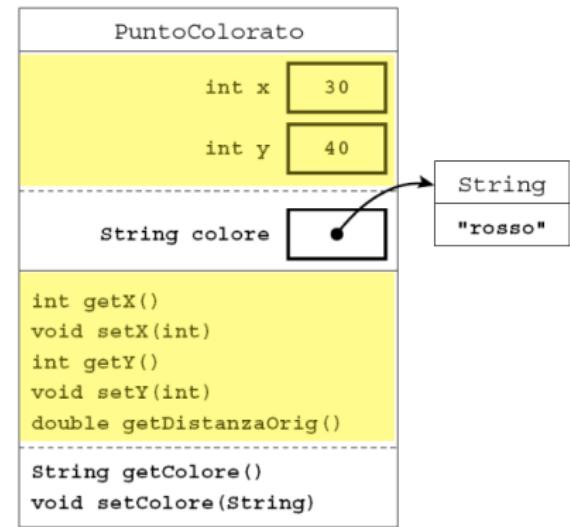
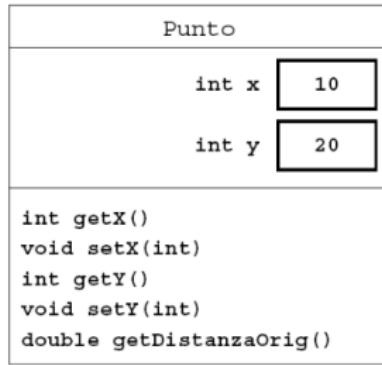
```
public class Punto {  
    private int x, y;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {return x;}  
    public int getY() {return y;}  
    public void setX(int newX) {x=newX;}  
    public void setY(int newY) {y=newY;}  
    public double getDistanzaOrig(){  
        return Math.sqrt(this.x * this.x +  
                        this.y * this.y);  
    }  
}
```

Ereditarietà: una sottoclasse di Punto

- PuntoColorato specializza Punto aggiungendo una variabile di istanza, colore, con i metodi accessori

```
public class PuntoColorato extends Punto {  
    private String colore;  
    public PuntoColorato(int x, int y) {  
        this(x, y, "nero");  
    }  
    public PuntoColorato(int x, int y, String colore) {  
        super(x, y);  
        this.colore = colore;  
    }  
    public String getColore() {  
        return colore;  
    }  
    public void setColore(String colore) {  
        this.colore = colore;  
    }  
}
```

Ereditarietà: struttura delle istanze



- Un'istanza di PuntoColorato è composta di due parti: quella ereditata da Punto, e quella nuova

Compatibilità tra classe e superclasse

- Se un metodo ha un parametro di tipo C, posso invocarlo passando come argomento un'istanza di una sottoclasse (diretta o indiretta) di C. Es:

```
public static void stampaDistanzaOrig( Punto p){  
    double dist = p.getDistanzaOrig();  
    System.out.println("Distanza dall'origine: " + dis);  
}
```

- Il codice seguente è corretto

```
Punto p = new Punto(30,40);  
stampaDistanzaOrig(p);  
  
PuntoColorato pc = new PuntoColorato(20,15,"rosso");  
stampaDistanzaOrig(pc);
```

L'operatore `instanceof`

- L'operatore **`instanceof`** serve per controllare il tipo dinamico di un'espressione; è un operatore relazionale

*EsprRel ::= Espressione **`instanceof`** TipoRif*

- *Espressione* è un'espressione che deve produrre un riferimento
- *TipoRif* è un nome di un tipo riferimento (array, classe o interfaccia) non parametrizzato
- La valutazione di

`espr instanceof Tipo`

restituisce **`true`** se `espr` è diversa da **`null`** e il tipo dinamico di `espr` è compatibile verso `Tipo`

L'operatore di cast

- Come nel caso dei tipi primitivi, il **cast** permette di indicare al compilatore che un'espressione deve essere considerata di un tipo diverso da quello statico. Sintassi:

EsprUna ::= (TipoRif) Espressione

- L'espressione

(Tipo) espr

dove *Tipo* è un tipo riferimento, compila correttamente se *Tipo* è un sottotipo o un sovratipo del tipo statico di *espr* (il caso interessante è essere sottotipo!)

- In questo caso, il compilatore assegna a *(Tipo) espr* il tipo statico *Tipo*
- Se a tempo di esecuzione il tipo dinamico di *espr* non è compatibile verso *Tipo*, verrà lanciata una eccezione di tipo *ClassCastException*

Cast: un esempio e esercizio

- Per prima cosa scriviamo il metodo `chiediPunto()` che restituisce o un Punto o un PuntoColorato le cui coordinate ed eventuale colore sono lette da console

```
public static ????. chiediPunto()
```

(cosa scrivo al posto di ?????).

- Assumiamo di avere a disposizione:

```
public static Punto leggiCoordinate()
```

che restituisce un Punto le cui coordinate ed eventuale colore sono lette da console.

```
Punto p = chiediPunto();
if (p instanceof PuntoColorato) {
    PuntoColorato pC = (PuntoColorato) p;
    String col = pC.getColore();
    System.out.println("Punto di colore: " + col);
} else {
    System.out.println("Punto scolorato!");
}
```

- L'eccezione `ClassCastException` non verrà mai lanciata perché il cast viene fatto dopo aver controllato il tipo dinamico di `p` con `instanceof`

Cast: compatibilità

- Il tipo statico dell'espressione di cui si fa il cast deve essere sovratipo o sottotipo del tipo a cui si fa il cast:

```
public class A {  
    public void m() {  
        A a;  
        B b;  
        A c;  
        a= new A();  
        b= new B();  
        c=(A) a;  
        c=(A) b;  
        c= (C) b; //Cannot cast from B to C  
    }  
}  
  
class B extends A {}  
class C extends A {}
```

- però si può fregare il compilatore!!!!

```
c = (C)((Object) b); // questo e' possibile
```

Cast di tipi primitivi e di tipi riferimento (confronto)

- Il cast tra tipi riferimento non modifica mai l'oggetto cui è applicato, es:

```
1 PuntoColorato pC = new PuntoColorato(5,10,"blu");
2 Punto p = pC; //cast automatico
3 PuntoColorato pC1 = (PuntoColorato) p; //cast esplicito
4 System.out.println(pC==pC1);
5 // true
```

Alla riga 4 viene stampato **true**

- Il cast tra tipi primitivi invece può modificare il dato, es:

```
1 int x = Integer.MAX_VALUE -1;
2 float f = x; //cast automatico
3 int y = (int) f; //cast esplicito
4 System.out.println("x: " + x + ", f: " + f + ", y: " + y);
5 //x:2147483646,f:2.14748365E9,y:2147483647
6 System.out.println(x==y);
7 // false
```

Alla riga 6 viene stampato **false!**

Overriding e binding dinamico

- L'ereditarietà consente di sovrascrivere, **override**, un metodo di una superclasse in una sua sottoclasse, dichiarando un metodo con firma identica. Es:

```
public class Punto {  
    /* .... */  
    public String toString(){  
        return "Punto x:"+x+", y:"+y;  
    }  
}
```

```
public class PuntoColorato extends Punto {  
    /* .... */  
    public String toString(){  
        return "Punto col. x:"+x+", y:"+y+", col:"+ colore;  
    }  
}
```

- Il metodo `toString()` è sovrascritto in `PuntoColorato` per indicare anche il colore.

Regole per la sovrascrittura

- Per sovrascrivere un metodo `met` di una superclasse in una sua sottoclasse, il metodo che lo sovrascrive deve avere
 - la stessa firma di `met` (nome e parametri di input)
 - il tipo del risultato deve essere uguale a quello di `met` se è un tipo primitivo, compatibile verso quello di `met` se è un tipo riferimento (sottoclasse)
 - avere una visibilità uguale o maggiore di quella di `met`

Regole di accessibilità per membri di classe: esempio

```
public class Punto {  
    private int x, y;  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public long maxX(Punto p) {  
        return (p.x > x) ? p.x : x;  
    }  
    long minX(Punto p) {  
        return (p.x > x) ? x : p.x;  
    }  
    Punto getPunto() {  
        return new Punto(2,3);  
    }  
}
```

```
public class PuntoColorato extends Punto {  
    private String colore;  
    public PuntoColorato(int x, int y, String colore) {  
        super(x,y);  
        this.colore = colore;  
    }  
    public long maxX (Punto p){ return 0; } //OK override  
  
    long maxX(Punto p){//ERRORE ne' override ne' overload  
        return 0;}  
  
    long maxX(int s) { return 0; } //OK overload  
  
    public long minX(Punto p) { //OK override  
        return 0;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Punto p;  
        PuntoColorato pc;  
  
        p = new PuntoColorato(1, 2, "rosso");  
        pc = new PuntoColorato(3, 4, "blue");  
  
        p.maxX(pc);  
        p.maxX(3); // ERRORE questo non e' definito per Punto  
        pc.maxX(3); // OK  
        ((PuntoColorato) p).maxX(3); // OK  
    }  
}
```

Il binding dinamico

- Quando si invoca un metodo d'istanza (sovrascritto in una o più classi), Java determina il metodo da eseguire a tempo di esecuzione, in base al tipo dinamico dell'espressione su cui è invocato. Es:

```
Punto p = chiediPunto();
System.out.println(p.toString());
```

- Viene eseguito il metodo `toString()` di `Punto` o di `PuntoColorato` a seconda del tipo dell'oggetto restituito da `chiediPunto()`.
- Notate che il compilatore **NON** può sapere quale metodo `toString()` verrà chiamato.
- Questo meccanismo si chiama **binding dinamico**. (In C++ e C# i metodi soggetti al binding dinamico devono essere dichiarati con modificatore `virtual`!)

Hiding di variabili e di metodi statici

- Cosa succede se in una sottoclasse dichiaro un membro con lo stesso nome di un membro ereditato? Dipende dal tipo di membro!
- Per i **metodi di istanza**, abbiamo visto l'**overriding**
- Una variabile/campo viene **nascosto** da una dichiarazione di variabile/campo con lo stesso nome (anche di tipo diverso) nella sottoclasse,
 - gli oggetti della sottoclasse avrebbero due variabili con lo stesso nome, di cui solo una visibile direttamente, **questa NON è una buona pratica!**
- Per i **metodi statici** può essere **nascosto** in una sottoclasse: la scelta del metodo da eseguire è gestita staticamente dal compilatore

Hiding di variabili: un esempio

- Se abbiamo la definizione della classe Punto e PuntoColorato che seguono

```
public class Punto {  
    private int x, y;  
    .....  
}  
  
public class PuntoColorato extends Punto {  
    private int x, y;  
    private String colore;  
    .....  
}
```

in un oggetto di tipo PuntoColorato ci sono 5 campi: 4 di tipo **int** e uno di tipo **String**.

- Chiaramente questo non è quello che aspettiamo!

Accesso alla superclasse con `super`

- Se un metodo (o una variabile) d'istanza della superclasse sono nascosti da una dichiarazione locale, gli si può accedere usando **super**

AccediCampo ::= **super**.*Id* | ...

RifMetodo ::= **super**.*ArgsDiTipo?* *Id* | ...

- Concettualmente, **super** denota l'istanza corrente (come **this**), ma vista come istanza della superclasse
 - Può essere usato solo con dot-notation (differentemente da **this**)
 - Non può essere iterato: **super.super.x** è un **ERRORE**

Uso di `super`

- `super` viene usato tipicamente per accedere a un metodo della superclasse non accessibile perchè sovrascritto. Una nuova versione del metodo `toString()` per le classi `Punto` e `PuntoColorato`

```
public class Punto {  
    /* .... */  
    public String toString(){  
        return "Punto x:"+x+",y:"+y;  
    }  
}
```

```
public class PuntoColorato extends Punto {  
    /* .... */  
    public String toString(){  
        return super.toString()+" col:"+ colore;  
    }  
}
```

- **ATTENZIONE:** Cosa succede se rimpiazziamo `super.toString()` con `toString()`?

Costruttori e ereditarietà

- Sappiamo che con **this**(...) si può invocare un diverso costruttore della classe corrente solo come primo comando di un costruttore
- Nella stessa posizione si può invocare anche un costruttore della superclasse, usando **super**(...)

*CorpoCostruttore ::= { InvocaCostruttore? CmsInBlocco**

InvocaCostruttore ::= ArgsDiTipo? (this | super) (Args?)

- anche **super**(...) viene valutato in contesto statico
- **this**(...) e **super**(...) non possono essere presenti entrambi
- Ogni costruttore che non ha come prima istruzione **this**(...) o **super**(...) invoca (implicitamente) il costruttore senza argomenti della superclasse, come se avesse **super**() come primo comando
 - Se la superclasse è Object, **super**() non fa nulla
 - Se la superclasse non ha un costruttore senza argomenti (nè esplicito nè default) ci sarà un errore di compilazione

Tipo statico e tipo dinamico di variabile

- Poiché il tipo di una classe è compatibile per assegnamento verso il tipo della sua superclasse, posso assegnare un PuntoColorato a una variabile di tipo Punto:

```
Punto p = new PuntoColorato(10,20) //OK
```

- Chiamiamo **tipo statico** di una variabile quello della sua dichiarazione, e **tipo dinamico** quello dell'oggetto ad essa assegnato
 - p ha tipo statico Punto tipo dinamico PuntoColorato
 - il tipo dinamico può variare durante l'esecuzione
 - Il tipo dinamico è sempre compatibile verso quello statico
- Attenzione: non posso fare il viceversa:

```
PuntoColorato p = new Punto(10,20) //ERRORE
```

Ereditarietà e invocazione di costruttori

- Quindi quando si valuta un'espressione **new C()** viene eseguito almeno un costruttore delle classe C e uno di ogni sua superclasse. L'ordine è discendente:

```
class A {  
    A(){System.out.println("Costruttore di A");}  
}  
class B extends A {  
    B(){System.out.println("Costruttore di B");}  
}  
public class C extends B {  
    C(){System.out.println("Costruttore di C");}  
    public static void main (String[] args){  
        C c = new C();  
    }  
}
```

- Cosa viene stampato quando eseguiamo **new C()**?

Costruttore di A
Costruttore di B
Costruttore di C

Ereditarietà e costruzione di istanze

- Valutazione di un costruttore in presenza di ereditarietà:
 - Se la prima istruzione è un'invocazione di **this(...)**:
 - si esegue il costruttore individuato dai parametri di **this(...)**
 - quando termina, si esegue il resto del costruttore
 - si termina restituendo un riferimento all'istanza creata
 - Altrimenti, la prima istruzione invoca (implicitamente o esplicitamente) un costruttore della superclasse (nel caso non ci sia una chiamata esplicita invoca **super()**):
 - si esegue il costruttore individuato dai parametri di **super(...)**
 - quando termina si esegue il resto del costruttore
 - si termina restituendo un riferimento all'istanza creata

La classe Object

- Object è la radice della gerarchia di ereditarietà:
 - tutte le classi Java ereditano (direttamente o indirettamente) da Object
 - Object fornisce una decina di metodi di utilità generale
 - Alcuni vengono ereditati e usati direttamente:
 - `wait()`, `notify()`, ..., usati per programmazione concorrente
 - `getClass()` restituisce l'oggetto di tipo `Class` che rappresenta la classe dell'oggetto
- Gli altri vengono normalmente ridefiniti nelle sottoclassi
 - `finalize()`, invocato dal garbage collector prima di deallocare
 - `clone()`, per creare una copia
 - `equals()` e
 - `toString()`

Il metodo equals()

- Sappiamo che ==, applicato a due espressioni di tipo riferimento, le confronta per identità: restituisce true solo se denotano lo stesso riferimento
- equals() confronta due oggetti per uguaglianza, e va sovrascritto in ogni classe per utilizzare i criteri di confronto adeguati alla classe
 - si scrive o1.equals(o2) per confrontare o1 e o2
 - deve soddisfare alcune proprietà descritte nelle API, tra cui **riflessività**, **simmetria** e **transitività**
- Nella classe Object, per oggetti diversi da **null**, o1.equals(o2) se e solo se o1 == o2

Metodo equals(): esempio

- Come scrivere un metodo equals(_) che funzioni correttamente con sottoclassi.

```
public class Punto {  
    public boolean equals(Object obj){  
        if ( obj==null || obj.getClass()!=this.getClass() ) {  
            return false;  
        }  
        Punto pun=(Punto) obj;  
        return ( this.x==pun.getX() && this.y==pun.getY() );  
    }  
    .....  
}  
  
public class PuntoColorato extends Punto{  
    public boolean equals(Object obj){  
        if (!super.equals(obj)){  
            return false;  
        }  
        PuntoColorato punC=(PuntoColorato) obj;  
        return ( this.colore.equals(punC.getColore()) );  
    }  
    .....  
}
```

Esecuzione equals(): esempio

- Come avviene la valutazione delle seguenti istruzioni (assumiamo che siano parte del metodo **public static void main(String[] s)** del programma).

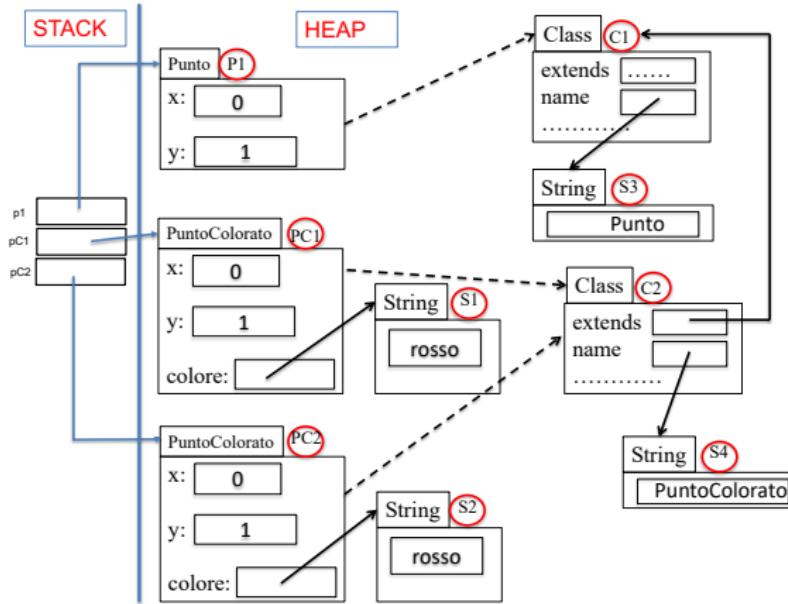
```
Punto p1 = new Punto(0,1);
PuntoColorato pC1 = new PuntoColorato(0,1,"rosso");
PuntoColorato pC2 = new PuntoColorato(0,1,"rosso");
boolean b1=pC1.equals(p1);
boolean b2=pC1.equals(pC2);
```

Esecuzione equals(): heap

- Eseguiamo

```
boolean b1=pC1.equals(p1);  
boolean b2=pC1.equals(pC2);
```

- L'heap dopo la creazione dei 3 punti contiene fra gli altri i seguenti oggetti, assumiamo che il riferimento all'oggetto sia indicato nel cerchio in rosso.



Il metodo `toString()`

- `toString()` restituisce una rappresentazione testuale dell'oggetto su cui è invocato
- In `Object` restituisce una stringa della forma `Classe@hashcode`, con la classe dell'oggetto e un codice esadecimale che lo identifica nella JVM
- Quindi va ridefinito, ed è opportunamente ridefinito in tutte le classi della distribuzione Java
 - Es: in `java.util.ArrayList`, `toString()` invoca se stesso su tutti gli oggetti del contenitore

Le classi astratte

- Una classe astratta è una classe dichiarata con il modificatore **abstract**
- Una classe astratta **può contenere metodi astratti**. Non deve necessariamente contenerne!
 - Un **metodo astratto** è un metodo in cui manca il corpo, sostituito da ; e qualificato dal modificatore **abstract**
- Se una classe contiene un metodo astratto, **deve** essere dichiarata con **modificatore abstract**.
- **IMPORTANTE:** Non si può fare **new** specificando una classe astratta! Queso perchè all'oggetto che si costruirebbe potrebbero mancare i body di alcuni metodi

Classi astratte ed ereditarietà

- Una classe **concreta** (non astratta) per estenderne una astratta deve fornire un'implementazione di tutti i suoi metodi astratti, cioè li deve sovrascrivere
- Rispetto alle classi concrete, l'unico vincolo di una classe astratta è che **non se ne possono creare istanze**
- A una variabile il cui tipo è una classe astratta si possono assegnare istanze delle sue sottoclassi concrete
- Le sottoclassi di una classe astratta condividono metodi (anche astratti) e variabili che ereditano da essa:
 - le classi astratte permettono di evitare ridondanza del codice anche quando le informazioni comuni a più classi non definiscono completamente una classe

Classi astratte: un esempio

- La classe Solido rappresenta corpi tridimensionali di materiale omogeneo
- È astratta perché il volume dipende dalla forma, non specificata a questo livello di astrazione
- A una variabile il cui tipo è una classe astratta si possono assegnare istanze delle sue sottoclassi concrete

```
public abstract class Solido{  
    private double pesoSpecifico;  
    public Solido(double pesoSpecifico){  
        this.pesoSpecifico = pesoSpecifico;  
    }  
    public double getPeso(){  
        return getVolume()*pesoSpecifico;  
    }  
    public abstract double getVolume();  
}
```

Classi astratte: classi concrete

- Sfera estende Solido fornendo un'implementazione di getVolume()

```
public class Sfera extends Solido{  
    private double raggio;  
    public Sfera(double raggio, double pesoSpecifico){  
        super(pesoSpecifico); // se non presente ERRORE!  
        this.raggio = raggio;  
    }  
    public double getVolume(){  
        return (4.0/3.0) * Math.PI * Math.pow(raggio,3);  
    }  
    public String toString(){  
        return "Sfera (" + raggio + ")";  
    }  
}
```

- Cubo estende Solido fornendo un'implementazione di getVolume()

```
public class Cubo extends Solido{  
    private double lato;  
    public Cubo(double lato, double pesoSpecifico){  
        super(pesoSpecifico); // se non presente ERRORE!  
        this.lato = lato;  
    }  
    public double getVolume(){  
        return Math.pow(lato,3);  
    }  
    public String toString(){  
        return "Cubo [" + lato + "]";  
    }  
}
```

Note su metodi e classi astratte

- Una classe può contenere un metodo astratto anche se non compare esplicitamente nel corpo:
 - se non sovrascrive un metodo astratto ereditato da una classe astratta che estende, direttamente o non
 - (come vedremo) se non sovrascrive un metodo astratto di una interfaccia che implementa, direttamente o non
- In questi casi la classe deve essere astratta
- Una classe astratta può estendere una classe concreta: può anche sovrascrivere un metodo concreto con uno astratto, anche se questo è abbastanza raro!

Corso: Paradigmi di Programmazione

Paola Giannini

Interfacce e Generici

Ereditare e implementare

- Sappiamo che una classe può estendere direttamente una sola superclasse, **ereditarietà singola**
 - l'ereditarietà singola non è del tutto soddisfacente
 - a seconda dell'uso che vogliamo fare di certi oggetti potrebbe essere conveniente avere viste diverse su di essi, a livelli di astrazione indipendenti o ortogonali rispetto alla gerarchia dell'ereditarietà
- Una classe può invece **implementare più interfacce**
 - questo offre una soluzione soddisfacente ai requisiti elencati sopra

Le interfacce

- Un'interfaccia ha una struttura simile a una classe astratta, ma i metodi che contiene o sono astratti o se hanno un body devono avere il modificatore **default**. Prima Java8 le interfacce potevano contenere SOLO metodi astratti.
- Le interfacce, come le classi, sono tipi
- Come per le classi astratte anche per le interfacce non si può creare un oggetto di questo tipo (fare le **new**)
- Una classe può dichiarare di implementare una o più interfacce:
 - deve fornire un'implementazione per tutti i metodi astratti delle interfacce che implementa (può non specificare l'implementazione dei metodi **default**)
 - oppure deve essere dichiarata astratta, perché è come se contenesse i metodi astratti che non implementa

Alcuni esempi di uso

- Come sviluppare algoritmi di ordinamento che non dipendano dalla natura specifica degli oggetti?
 - ogni algoritmo si applica a elementi **confrontabili** tra loro anche se non raggruppabili sfruttando l'ereditarietà, ad esempio: String, Integer, Double o altri ancora
 - Uso dell'interfaccia **Comparable<T>**
- Analogamente: come trattare in modo uniforme collezioni **iterabili** di oggetti?
 - scorrere gli elementi della collezione
 - controllare se la collezione è vuota,
 - Uso dell'interfaccia **Iterable<T>**

La dichiarazione di interfacce

- Ogni interfaccia introduce un nuovo tipo riferimento di cui ne sono istanze tutti gli oggetti delle classi che implementano l'interfaccia

DichInterfaccia ::= *Modificatore** **interface** *Id ParamsDiTipo*?
EstendeInterfaccia? *CorpolInterfaccia*

EstendeInterfaccia ::= **extends** *IdQArgs*

IdQArgs ::= *IdQArg* (, *IdQArg*)*

IdQArg ::= *Id ArgDiTipo*?

CorpolInterfaccia ::= { *DichInInterfaccia** }

DichInInterfaccia ::= *DichCampo* | *DichMetodo* | *DichClasse* | *DichInterfaccia*

- Come evidente dalle produzioni, la struttura di un'interfaccia è simile a quella di una classe

- la parola chiave **interface** rimpiazza **class**

- manca la clausola *Implements*

- il corpo può contenere solo dichiarazioni di classi, interfacce, campi costanti e metodi astratti

- Un'interfaccia è detta interfaccia membro se dichiarata all'interno di un'altra interfaccia o classe, top-level altrimenti

Modificatori e membri

- I modificatori consentiti per un'interfaccia sono le annotazioni, **public**, nessun modificatore, **protected**, **private**, **static**
 - ogni interfaccia è implicitamente **abstract**
 - per il momento consideriamo solo interfacce **public** e top-level dichiarate in un file che ha lo stesso nome dell'interfaccia e suffisso **.java**
- Membri
 - tutti i membri sono dichiarati implicitamente come **public**
 - i campi sono dichiarati implicitamente **static** e **final**, definiscono delle costanti e devono essere accompagnati da un'espressione di inizializzazione
 - tutti i metodi sono implicitamente **abstract** (non si deve specificare il modificatore)

Esempi di interfacce

- Pesabile è un'interfaccia che specifica un solo metodo getPeso() e viene usata come tipo nel metodo carica della classe Ascensore nel quale viene calcolato se quello che è caricato nell'ascensore ha un peso inferiore al peso limite caricabile

```
public interface Pesabile{
    double getPeso(); //e' implicitamente public
}
public class Contenitore {
    public final double LIMITE; // in grammi
    private double pesoContenuto;
    private ArrayList<Pesabile> contenuto;
    //Costruttore e getter pesoContenuto

    public void aggiungi(Pesabile obj) {
        double temp = pesoContenuto + obj.getPeso();
        if (temp > LIMITE) {
            System.out.println("Peso eccessivo");
        } else {
            contenuto.add(obj);
            pesoContenuto = temp;
        }
    }
}
```

Interfacce nelle librerie

- Le librerie di Java contengono molte interfacce utili. Ad esempio:

```
package java.lang;
.....
public interface CharSequence{
    // length() returns the length of this character sequence
    int length();
    // charAt(n) returns the char value at index n
    char charAt(int index);
    // subSequence(n,m) returns a new CharSequence that is the
    // subsequence of this sequence from index n (included) to
    // m (excluded)
    CharSequence subSequence(int start, int end);
    // toString() returns a string containing the characters in
    // this sequence in the same order as this sequence
    String toString();
}
```

- String, StringBuffer e StringBuilder implementano questa interfaccia.
- Potete scrivere un metodo che fa lo startsWith fra sequenze

```
public static boolean startsWith (CharSequence pr,CharSequence str)
```

che funziona per input di tipo String, StringBuffer e StringBuilder.

Implementare le interfacce

- Data l'interfaccia CharSequence
 - possiamo dichiarare variabili di tipo CharSequence
 - possiamo invocare su di queste i metodi definiti nell'interfaccia CharSequence possiamo definire metodi che hanno argomenti di tipo CharSequence o che restituiscono valori di tipo CharSequence
- Ma, come per le classi astratte non possiamo creare un oggetto di tipo CharSequence con la **new**.
- Dobbiamo definire una “classe concreta” che:
 - implementi tutti i **metodi astratti** dell'interfaccia (definisca un body)
 - l'intestazione della classe deve **dichiarare** in modo esplicito quali **interfacce implementa**. In questo modo il compilatore assicura la corrispondenza tra i nomi dei metodi della classe e quelli delle interfacce.

Esempi

- Esempio 1: Usiamo il metodo aggiungi della classe Contenitore. Cosa possiamo mettere nel Contenitore?
- Esempio 2: Attenzione ai campi delle interfacce!

```
public interface PesabileCs {  
    double peso=3; //e' implicitamente public static final  
    double peso(); //e' implicitamente public e abstract  
}
```

Rispettare le specifiche

- La realizzazione di un metodo deve sempre rispettare la descrizione testuale (**specifica**) che accompagna ogni metodo astratto
- Con **Java 8**, in un'interfaccia è possibile specificare per un metodo la sua implementazione, tramite la parola chiave **default**, definendo il comportamento di default. Es:

```
public interface Interfaccia1 {  
    default void printMessage(String str){  
        System.out.println("Interfaccia 1: " + str);  
    }  
}  
public class DefaultMethodExample implements Interfaccia1{}
```

- L' implementazione di default può essere ridefinita se si vuole reimplementare il metodo.

La gerarchia di interfacce e classi

- La gerarchia di ereditarietà singola delle classi e quella multipla delle interfacce **devono intendersi come relazioni disgiunte**
- Però si noti che se una classe `Cla` implementa un'interfaccia `Inte`, allora:
 - tutte le sottoclassi di `C` implementano `I`
 - tutte le superinterfacce di `I` sono implementate da `C` (e dalle sue sottoclassi)

Dichiarazione di classi generiche

- Ritorniamo alle dichiarazioni di classe

DichClasse ::= Modificatore **class** Id ParamsDiTipo?
Estende? Implementa? CorpoClasse*

- Come per i metodi generici, *ParamsDiTipo* è una sequenza di variabili di tipo, eventualmente con vincoli

ParamsDiTipo ::= < ParamDiTipo (, ParamDiTipo) >*

*ParamDiTipo ::= Id (**extends** TipoRif (& TipoRif)*)? | ...*

- Se *ParamsDiTipo* è presente, la classe è **generica**.

Intestazione di una classe generica

- L'intestazione della classe generica `ArrayList<E>` del package `java.util` contiene tutte le componenti opzionali

```
public class ArrayList <E>
    extends java.util.AbstractList <E>
    implements Serializable, Cloneable,
               Iterable <E>, Collection <E>, List <E>,
               RandomAccess
{
    /* corpo della classe */
}
```

- Notate la classe
 - ➊ è generica,
 - ➋ estende una classe (probabilmente dal nome astratta!) generica e
 - ➌ implementa molte interfacce fra le quali delle interfacce generiche!

Interfacce generiche e superinterfacce

- Come per le classi, un'interfaccia può avere dei parametri di tipo; in questo caso
 - viene detta **interfaccia generica**
 - lo scope dei parametri di tipo si estende dal punto in cui sono dichiarati fino alla fine del corpo dell'interfaccia
- Un'interfaccia **non può estendere una classe**
- Un'interfaccia può estendere una o più interfacce
 - le interfacce estese (direttamente o indirettamente) da un'interfaccia sono le sue **superinterfacce**
 - un'interfaccia eredita tutti i metodi astratti delle sue superinterfacce

Metodi generici

- Abbiamo visto finora come i metodi possano prendere dei **valori** come argomenti da associare ai **parametri**. Ad esempio
 - parametri formali: $T \ m(T_1 \ id_1, T_2 \ id_2)$
 - parametri attuali (argomenti): $m(v_1, v_2)$
- Java permette di definire metodi che prendano dei **tipi** come **parametri**. Ad esempio
 - parametri di tipo formali: $\langle id_a, id_b \rangle \ id_a \ m(T_1 \ id_1, id_b \ id_2)$
 id_a e id_b possono essere usati nel body di m come se fossero tipi normali
 - parametri di tipo attuali: $\langle T_a, T_b \rangle \ m(v_1, v_2)$
i tipi T_a e T_b devono essere **tipi riferimento**
- Si parla in questo caso di **metodi generici** in cui la genericità si riferisce ai tipi

Metodi generici: sintassi definizione

- La dichiarazione di metodo quindi può specificare parametri di tipo:

DichMetodo ::= Modificatore ParamsDiTipo? Tipo Id (Params) CorpoMetodo*

ParamsDiTipo ::= < ParamDiTipo (, ParamDiTipo) >*

ParamDiTipo ::= Id (extends TipoRif (& TipoRif))?*

- Nel caso più semplice, un *ParamDiTipo* è semplicemente un *Id*. Per convenzione, si usano singole lettere maiuscole.
- Un *ParamDiTipo* può anche assumere la forma più complessa che analizzeremo quando vedremo le classi generiche.

Esempio metodo generico

Scriviamo un metodo che fa il swap di due elementi di un array:

i parametri dovranno essere l'array e i due indici (interi) (in caso uno dei due indici sia fuori dall'array non si deve modificare l'array).

```
public static <T> void swap(T [] array, int i, int j)
```

Metodi generici: sintassi chiamata

- Abbiamo visto la sintassi del *RifMetodo* per metodi non generici. La sintassi che include anche i generici è come segue:

```
RifMetodo ::= Id | ( Id. )+ ArgDiTipo? Id | ExprPri . ArgsDiTipo? Id  
ArgsDiTipo ::= < ArgDiTipo ( , ArgDiTipo )* >  
ArgDiTipo ::= TipoRif
```

- prima del nome del metodo possono essere specificati gli argomenti di tipo che **devono essere tipi riferimento** racchiusi da parentesi angolate.
- Una chiamata del metodo definito nella pagina precedente può essere

```
Integer a1 []= {0,1,2,3};  
.....  
<Integer>swap(a1, 1, 3);
```

Java però permette anche la chiamata semplice

```
swap(a1, 1, 3);
```

perchè può inferire il tipo dal contesto!

Metodi generici (con parametri di tipo vincolato)

- Abbiamo già visto i metodi generici con parametri semplici, ora vediamo quelli con parametri vincolati!
- Un *ParamDiTipo* può anche assumere la forma più complessa

Id **extends** C & I1 & I2 ...

dove C è una classe e I1, I2 ,..... sono interfacce

- In questo caso, sono accettabili come parametri attuali solo tipi che estendano la classe e implementino le interfacce indicate.
- Vogliamo scrivere un metodo max che prenda un numero variabile di argomenti e sia generico nel tipo degli elementi, cioè funzioni per tutti i T che sono **confrontabili**
- Per confrontare oggetti abbiamo bisogno che la loro classe abbia un metodo che permette di fare il confronto di fra un oggetto della classe (**this**) e il parametro del metodo
- L'interfaccia Comparable<T> contiene metodo (astratto) **int compareTo(T el)** che ritorna
 - un valore positivo se **this** è maggiore di el
 - un valore negativo se **this** è minore di el
 - 0 se **this** è uguale a el

Il metodo max

```
public static <T extends Comparable<T>> T max(T...n){  
    if (n.length==0) return null;  
    T max=n[0];  
    for (T i:n) if (i.compareTo(max)>0) max=i;  
    return max;  
}
```

- T può essere qualunque tipo, purché
 - estenda Comparable<T>
 - e quindi implementi il metodo **int compareTo(T el)**
- Gli argomenti e il valore restituito sono di tipo T
- All'interno del corpo usiamo variabili di tipo T

Metodi generici: invocazione (1)

Alcuni esempi di invocazione:

- `int r = max (1,2,3);`

il tipo T = Integer (non `int`) viene inferito dal tipo degli argomenti (`int`) e poi promosso con boxing (Integer)

- `String s = max ("borsa", "abaco", "zaino");`

analogamente, T viene inferito dal tipo degli argomenti come String

Metodi generici: invocazione (2)

Possiamo definire la classe Voto che implementa Comparable<Voto> da usare come tipo dei parametri di max.

```
public class Voto implements Comparable<Voto> {
    private int voto;
    public Voto(int voto) {
        this.voto = voto;
    }
    public int getVoto() { return voto; }
    public int compareTo(Voto v) {
        if (voto < v.voto) return -1;
        else if (voto > v.voto) return +1;
        else return 0;
    }
    public String toString() {
        return "Voto: " + voto;
    }
}
```

e chiamare il metodo max

```
Voto v= max(new Voto(30), new Voto(18));
```



Costruttori generici

- Un **costruttore è generico** se il nome è preceduto da una lista di parametri di tipo, tra parentesi angolate (come i metodi)
- Considerazioni analoghe ai metodi generici:
 - un costruttore può essere generico indipendentemente dal fatto che la classe sia generica (come per i metodi)
 - quando viene invocato (con una **new**) si può fornire un tipo riferimento da associare al parametro di tipo, oppure no: in quest'ultimo caso il sistema inferisce (se possibile) il tipo corretto

Costruttori generici: un esempio

- Definizione di costruttori generici

```
public class Box<E> {  
    private E elemento;  
    public static String ultimaNota;  
    public Box(E elemento) {  
        this.elemento = elemento;  
    }  
    public <T> Box(E elemento, T arg) {  
        this(elemento);  
        ultimaNota= (arg==null)?null:arg.toString();  
    }  
    /* ..... */  
}
```

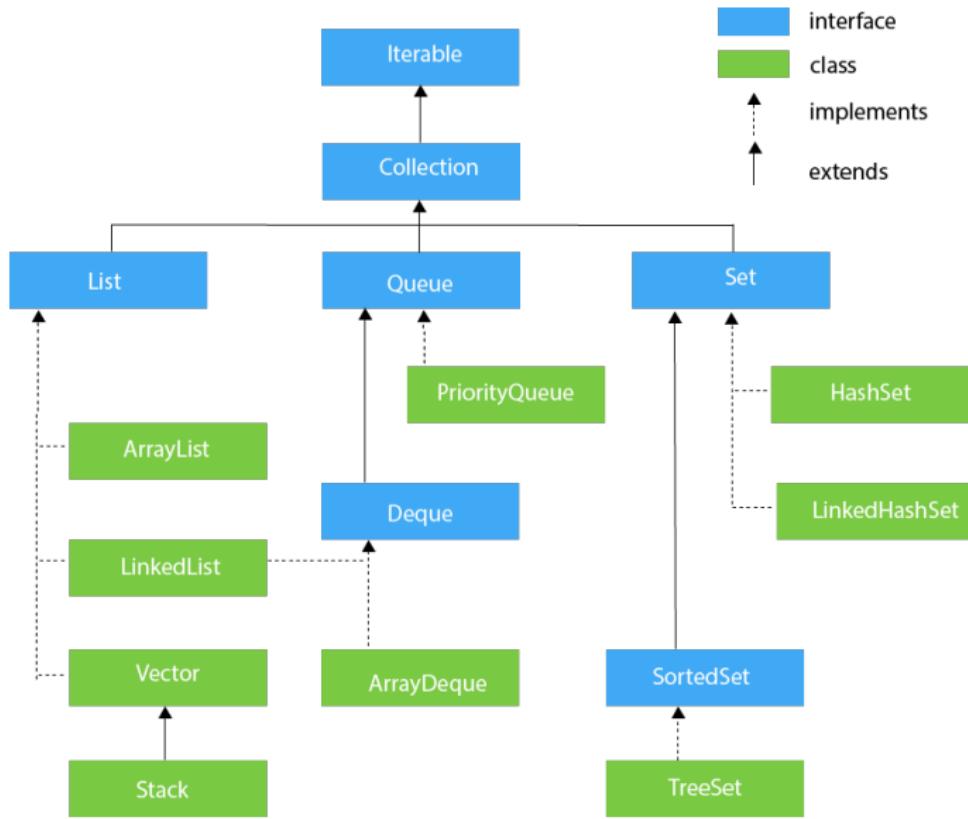
- Uso di costruttori generici

```
Box<Double>d=new <String> Box<Double>(3.14,"Ho messo un double");  
Box<Integer>i1= new Box<Integer>(3, "Ho messo un int");  
Box<Integer>i2= new Box(4, "Ho messo un int");
```

nota nel secondo e terzo caso il tipo String e poi anche Integer vengono inseriti

- Java Collections Framework (JCF)
 - è una architettura software
 - per la rappresentazione e manipolazione di gruppi di oggetti (le collezioni)
 - in maniera indipendente dall'effettiva realizzazione
- JCF comprende molte interfacce
 - la maggior parte dedicate a modellare collezioni con specifiche caratteristiche ed operazioni (ovvero Tipi di Dato Astratto)

Interfacce e classi (concrete) di JCF



Vediamo alcuni usi delle interfacce

- Come **Tipi di Dato Astratto (TDA)**. Specifica di operazioni (costruttori, selettori e test per strutture dati).
- Come **Contratti (CTR)**. Specifica di cosa si deve implementare per ottenere un certo servizio.
- Come **Proprietà (PROP)** che possono essere usate.

java.util.Collection<T> - TDA

- L'interfaccia più generale è Collection<T>: rappresenta un Tipo di Dato Astratto (TDA) **collezioni di oggetti**
 - che siano **omogenee**,
 - il parametro di tipo T indica il tipo degli oggetti ammessi nella collezione
 - senza ipotesi riguardo a:
 - l'organizzazione degli oggetti e la presenza di duplicati
 - la possibilità di aggiungere o rimuovere elementi
 - con operazioni essenziali:
 - isEmpty(), size(), contains(_), toArray()
 - sono invece opzionali i metodi add(_), remove(_), clear()

java.util.List<T> - TDA

- L'interfaccia List estende Collection assumendo che gli elementi siano ordinati in sequenza (non come valori)
 - i metodi che vengono aggiunti in List<T> riguardano l'**accesso posizionale**
 - `get(_)`, `indexOf(_)`, `lastIndexOf(_)`, `subList(_,_)`
 - e aggiunte, modifiche e rimozioni localizzate
 - `add(_,_)`, `set(_,_)`, `remove(_,_)`
- Le classi LinkedList, ArrayList e Vector implementano List e quindi anche Collection

- L'interfaccia Set estende Collection assumendo che il contenitore **non contenga elementi duplicati**.
- In particolare, NON ci sono elementi e1 e e2 tali che e1.equals(e2), e al più un elemento null
- La classe HashSet implementa l'interfaccia Set
- L'interfaccia SortedSet estende Set e gli elementi sono mantenuti in modo **ordinato** (implementano l'interfaccia Comparable)
- Infine TreeSet implementa l'interfaccia SortedSet

java.util.Set<T> un esempio

```
1 import java.util.*;
2 public class SetDemo {
3     public static void main(String args[]) {
4         List<Integer> list= Arrays.asList(34, 22, 10, 60, 30, 22, 34);
5         System.out.println("The list is:");
6         System.out.println(list);
7
8         Set<Integer> set = new HashSet<Integer>(list);
9         System.out.println("The set is:");
10        System.out.println(set);
11
12        SortedSet<Integer> sortedSet = new TreeSet<Integer>(set);
13        System.out.println("The sorted set is:");
14        System.out.println(sortedSet);
15        System.out.println("The first element of the set is: " + sortedSet.first());
16        System.out.println("The last element of the set is: " + sortedSet.last());
17
18        SortedSet<Integer> sortedSet1 = new TreeSet<Integer>(
19                                Arrays.asList(34, 22, 10, 60, 30, 22, 34));
20        System.out.println("The list sorted and is:");
21        System.out.println(sortedSet1);
22    }
23}
```

- riga 8 Set<T> è un'interfaccia implementata da HashSet<T>
- riga 12 e 18 SortedSet<T> è un'interfaccia implementata da TreeSet<T> SortedSet<T>
- i metodi first e last sono metodi di SortedSet<T> ma non di Set<T>

Iterable<T> e Iterator<T> - CTR

- Le interfacce possono stabilire dei CONTRATTI!
- L'interfaccia Iterable consente **per contratto** di applicare il **ciclo for-each** alle istanze delle classi che la implementano
 - dato che Collection<T> estende Iterable<T> , tutte le collezioni del JCF sono iterabili col for-each

```
public interface Collection<E> extends Iterable<E>
```

- L'interfaccia Iterable<T> comprende un solo metodo
 - iterator() che restituisce un oggetto di tipo Iterator<T>, chiamato iteratore .

Iterable<T> e Iterator<T> - CTR

- L'interfaccia Iterator offre un modo standard per scorrere collezioni di elementi
 - indipendentemente dalla loro natura
 - basato su due operazioni fondamentali e una opzionale:
 - la presenza di ulteriori elementi da analizzare (`hasNext()`)
 - la restituzione del riferimento a uno dei prossimi elementi non ancora analizzati (`next()`).
- La specifica del metodo `next()` prevede il lancio di un'eccezione di tipo `NoSuchElementException` se non ci sono altri elementi da analizzare
- opzionalmente, `remove()` rimuove l'ultimo elemento restituito

LinkedList<T> implementa Iterable<T>

```
public interface Iterable<T> {
    // Returns an iterator over a set of elements of type T.
    Iterator<T> iterator();
}

public interface Iterator<T> {
    // Returns true if the iteration has more elements.
    boolean hasNext()
    // Returns the next element in the iteration
    E next()
    // Removes from the underlying collection the last element returned by
    //this iterator (optional operation).
    void remove()
}
```

```
List<String> names = new LinkedList<String>();
// inserisci nomi

Iterator i = names.iterator();
while (i.hasNext()) {
    String name = (String) i.next();
    System.out.println(name);
}
// Il precedente ciclo while e' equivalente a
for (String name : names)
    System.out.println(name);
```

TDA e CTR: un esempio

- Considera il metodo generico che riceve due collezioni (come input) e restituisce un booleano che indica se le due collezioni sono disgiunte o no.

```
public static <T> boolean disgiunti(Collection<T> c1,  
                                     Collection<T> c2) {  
  
    for (T el : c1) {  
        if (!c2.contains(el))  
            return false;  
    }  
    return true;  
}
```

- Usa il TDA Collection con il metodo contains
- Usa il CTR Iterable per scorrere gli elementi con il for-each

Comparator<T>

- In aggiunta all'interfaccia Comparable esiste anche l'interfaccia Comparator

```
public interface Comparator <T>{  
    int compare (T obj1, T obj2);  
}
```

- La classe che implementa l'interfaccia Comparator è una classe che non ha dati, ma specifica che si deve fornire un modo per confrontare due oggetti.
- Il vantaggio di usare una classe Comparator è che si possono specificare più comparatori e si può utilizzare il comparatore più adatto al momento

Comparable<T> e Comparator<T>: un esempio

```
public class Persona implements Comparable<Persona>{
    private String nome;
    private String cognome;

    public int compareTo(Persona p) {
        return cognome.compareTo(p.cognome);
    }

    public String getNome(){return nome;}
    public String getCognome(){return cognome;}
}
```

- La classe Persona implementando Comparable fornisce l'**ordinamento naturale**, mentre con Comparator se ne possono fornire altri!

```
import java.util.Comparator;
public class ComparaPerNome implements Comparator<Persona> {
    public int compare(Persona a, Persona b) {
        return ((a.getNome()).compareTo(b.getNome()));
    }
}
```

Cloneable e Serializable - PROP

- L'interfaccia `java.lang.Cloneable` indica se sia lecito o meno che il metodo `clone()` della classe `Object` sia invocato sull'oggetto per farne una copia
- L'interfaccia `java.io.Serializable` indica se sia lecito o meno produrre una **rappresentazione binaria** dell'oggetto (es: per essere salvata sul file system)
- A puro titolo esemplificativo menzioniamo che:
 - `ArrayList` implementa entrambe
 - `Integer` implementa `Serializable` ma non `Cloneable`

Supertipo e sottotipo

- Interfacce e classi sono tipi. In generale si parla di supertipo e sottotipo
 - Il tipo **Object** è il supertipo di tutti i tipi
- Le relazioni di sottotipo e supertipo possono essere dirette o indirette
 - Una classe è sottotipo diretto di tutte le interfacce che dichiara di implementare
 - Una interfaccia è supertipo diretto di tutte le interfacce che dichiarano di estenderla
 - Se una classe S estende la classe C che implementa l'interfaccia I, allora S è sottotipo indiretto di I
- Ogni tipo è compatibile per assegnamento verso tutti i suoi supertipi, diretti e indiretti

Array, ArrayList e gerarchia dei tipi

- La gerarchia si estende agli array in modo intuitivo
 - Esempio: il tipo Number [] è supertipo di Integer []
 - Esempio: il tipo Object [] è supertipo di tutti i tipi array

```
Integer [] arrInt1 = new Integer [10];
Number [] arrNum = arrInt1; // OK
Integer [] arrInt2 = arrNum; // ERRORE
```

- La gerarchia però non si estende ai tipi parametrizzati

```
ArrayList<Integer> alstInt1 = new ArrayList<Integer>();
ArrayList<Number> alstNum = alstInt1; // ERRORE
ArrayList<Integer> alstInt2 = alstNum; // ERRORE
```

- I parametri di tipo sono solo annotazioni che consentono al compilatore di migliorare la precisione dell'analisi statica.
 - Per garantire la compatibilità all'indietro, a tempo di esecuzione (nella JVM) i tipi generici non sono più disponibili.
 - Il compilatore genera il codice scartando tutte le informazioni sui generici, sostituendo tutte le occorrenze dei parametri formali con Object (o col supertipo opportuno) e inserendo il cast esplicito dove necessario.
- La rimozione delle informazioni relative ai parametri di tipo è chiamata **erasure**.

Raw types

- A tempo di esecuzione esiste quindi una sola versione della classe generica
 - Es: nella fase di esecuzione esiste solo la classe `ArrayList` per tutti gli oggetti che nel codice avevano tipi parametrizzati quali `ArrayList<?>`, `ArrayList<String>`, `ArrayList<ArrayList<Integer>>`, `ArrayList<Object>`.
- Per garantire la compatibilità all'indietro, Java permette anche di scrivere programmi che usano tipi generici senza istanziarne i parametri. In questo caso si parla di **raw type**.
 - Es: possiamo definire e usare dati di tipo `ArrayList`.

Corso: Paradigmi di Programmazione

Paola Giannini

Classi Membro o Annidate

Classificazione di Classi e Interfacce

Le classi Java possono essere distinte in

- ① Classi **top-level**: quelle viste fino ad ora
- ② Classi **membro**: dichiarate come membro di un'altra classe
- ③ Classi **anonime**: dichiarate in un'espressione **new** (e che non hanno un nome)

Le prime due classificazioni si applicano anche alle Interfacce.

Le Classi Membro o Annidate

- Le classi viste finora sono classi **top-level**
- In Java possiamo scrivere più classi top-level in uno stesso file, ma solo una di esse può essere **public**, e questa deve avere lo stesso nome del file.
 - Questo può essere utile per limitare il numero di file, ma non introduce concetti nuovi
- Una classe è **membro/annidata** se la sua definizione compare all'interno di un'altra classe
- Anche se è possibile, raramente si usano classi annidate a più di un livello
- La relazione tra la classe membro e quella che la contiene non è banale, e dipende dal tipo di classe

Sintassi della classi membro

- Una **classe membro/annidata** è una classe dichiarata come membro di un'altra classe, cioè:

CorpoClasse ::= { *DichInClasse** }

DichInClasse ::= *DichCampo* | *DichCostruttore* | *DichMetodo* | *DichClasse*
| *DichInterfaccia* | *DichEnum* | *DichIniClasse* | *DichIniStanza*

- Una **classe membro è statica**, se la sua dichiarazione ha il modificatore **static**.

Le classi membro statiche

- Una classe membro **statica** Member dichiarata in una classe Outer differisce da una classe top level solo per aspetti di visibilità:
 - può avere anche il modificatore **protected** o **private**, con significato usuale
 - dall'esterno di Outer può essere riferita con il nome Outer.Member
 - Member può usare i membri statici di Outer con il loro nome semplice
 - Outer vede i membri di Member **anche se privati**

L'unica differenza fra le **classi membro statiche e quelle create al top-level** è il nome della classe, che nel primo caso è costituito dal nome della classe che la contiene “punto” il suo nome locale.

Classi membro statiche: definizione

```
1 public class Container {  
2     public static class Item {  
3         private Object data;  
4         public Item(Object data) {  
5             this.data = data;  
6         }  
7         public String toString() {return data.toString(); }  
8     }  
9     private ArrayList<Item> contenitore;  
10    public final String NOME;  
11    public Container(String nome) {  
12        contenitore = new ArrayList<Item>();  
13        NOME = nome;  
14    }  
15    public Item creaItem(Object data) {return new Item(data);}  
16    public Container addToContainer(Item item) {  
17        contenitore.add(item);  
18        return this;  
19    }  
20    public Container remFromContainer(Item item) {  
21        contenitore.remove(item);  
22        return this;  
23    }  
24    public String toString() {return NOME + contenitore.toString();}  
25 }
```

Dalla riga 2 alla 8 è definita la classe membro Container.Item

Classi membro statiche: uso

```
1 Container c1 = new Container("Amici");
2 Container c2 = new Container("Colleghi");
3 Container.Item it1 = new Container.Item("Pluto");
4 Container.Item it2 = c2.creaItem("Paperino");
5 Container.Item it3 = c1.creaItem("Pippo");
6 Container.Item it4 = new Container.Item("Nonna Papera");
7 System.out.println(c1.addToContainer(it1).addToContainer(it2).toString());
8 System.out.println(c2.addToContainer(it3).addToContainer(it2).addToContainer(it1).toString());
9 System.out.println(c2.remFromContainer(it3).toString());
10 System.out.println(c2.addToContainer(it4).toString());
```

risultato

```
Amici[Pluto, Paperino]
Colleghi[Pippo, Paperino, Pluto]
Colleghi[Paperino, Pluto]
Colleghi[Paperino, Pluto, Nonna Papaera]
```

- le righe 3 e 6 mostrano come creare un oggetto di tipo `Item` usando il suo costruttore
- alle righe 4 e 5 vengono creati oggetti di tipo `Item` che sono ritornati dal metodo `creaItem` della classe `Container`. È irrilevante se vengono creati da `c1` o da `c2`.
- Le righe 7-10 mostrano come usare i metodi **in catena** (**chainable**). Osservate che i metodi `addToContainer` e `remFromContainer` ritornano **this!** In questo modo si possono fare più operazioni in sequenza.

Le classi membro non statiche o classi interne

- Le classi membro non statiche sono chiamate anche **classi interne**
- Un'istanza di una classe interna (Inner) deve essere associata a un'**istanza della classe che la contiene** (Outer): questa può essere indicata come prefisso dell'espressione **new**

*EsprCreIstanza ::= (EsprPri.)? new ArgsDiTIp? IdQ (Args?)
CorpoClasse?*

- Ad esempio:

```
Outer.Inner istanzaInner=istanzaOuter.new Inner(...)
```

- Inner può accedere a membri d'istanza dell'associata istanza di Outer
- Inner **non può contenere membri statici**

Esempio di classe interna: iteratore

- Un **iteratore** è un oggetto che ci consente di visitare sequenzialmente tutti gli elementi di una collezione di dati usando due metodi:
 - `hasNext()` che restituisce **true** se c'è ancora un elemento da visitare
 - `next()` che restituisce il prossimo elemento ed avanza il cursore sulla struttura.
- Le classi Java che rappresentano collezioni di dati (sottotipi di `Collection`) offrono il metodo d'istanza
 - `iterator()`: che restituisce un iteratore sulla collezione
 - l'iteratore è un oggetto di una classe interna

Esempio di classe interna: iteratore (1)

Gli oggetti della classe `ArrayInteri` encapsulano un array di interi. Il costruttore genera una array di interi di dimensione `size` che contiene interi casuali da 0 a `bound`.

```
public class ArrayInteri {  
    private int[] arr;  
    public ArrayInteri(int size, int bound) {  
        arr = new int[size];  
        for (int i = 0; i < size; i++) {  
            arr[i] = (int) (Math.random() * bound);  
        }  
    }  
    ....  
}
```

Definiamo un iteratore sull'array di interi.

Esempio di classe interna: iteratore (2)

```
public class Iteratore implements Iterator<Integer> {
    // Indice dell'elemento corrente
    private int indiceCorrente;
    // Ritorna un iteratore in cui il primo elemento da restituire e'
    // quello nella prima posizione
    public Iteratore() {
        indiceCorrente = 0;
    }
    // Controlla se ci sono altri elementi da restituire
    public boolean hasNext() {
        return indiceCorrente < arr.length;
    }
    // Restituisce un elemento e si sposta in avanti
    public Integer next() {
        return arr[indiceCorrente++];
    }
}
```

- Questa è la classe interna (notate NON **static**) a ArrayInteri che ci permette di visitare gli elementi dal primo all'ultimo implementa un iteratore che produce interi e lo dichiariamo esplicitamente.
- Notate che ha una memoria interna (la variabile `indiceCorrente`) che ci dice quale è l'indice prossimo elemento restituito da `next()`

Esempio di classe interna: un altro iteratore (3)

```
public class IteratoreRev implements Iterator<Integer> {
    // Indice dell'elemento corrente
    private int indiceCorrente;
    // Ritorna un iteratore in cui il primo elemento da restituire e'
    // quello nell'ultima posizione
    public IteratoreRev() {
        indiceCorrente = arr.length - 1;
    }
    // Controlla se ci sono altri elementi da restituire
    public boolean hasNext() {
        return indiceCorrente >= 0;
    }
    // Restituisce un elemento e si sposta indietro
    public Integer next() {
        return arr[indiceCorrente--];
    }
}
```

- Anche questa classe interna definisce un iteratore che permette di visitare gli elementi dall'ultimo al primo.
- Queste due classi possono coesistere e si possono usare entrambe.

Esempio di classe interna: iteratore (3)

- Come si usa Iteratore per fare cicli annidati, cioè per ogni elemento di arrayInt vogliamo stampare tutti gli elementi di arrayInt

```
1 System.out.println("\nPrima stampa cicli annidati");
2 Iteratore it1 = arrayInt.new Iteratore();
3 Iteratore it2 = arrayInt.new Iteratore();
4 while (it1.hasNext()) {
5     System.out.print(it1.next() + ": ");
6     while (it2.hasNext()) {
7         System.out.print(it2.next() + "  ");
8     }
9     System.out.print("\n");
10 }
11
12 System.out.println("\nSeconda stampa cicli annidati corretta");
13 Iteratore it3 = arrayInt.new Iteratore();
14 while (it3.hasNext()) {
15     System.out.print(it3.next() + ": ");
16     ArrayInteri.Iteratore it4 = arrayInt.new Iteratore();
17     while (it4.hasNext()) {
18         System.out.print(it4.next() + "  ");
19     }
20     System.out.print("\n");
21 }
```

- Cosa viene stampato dalle righe 2-10 e dalle righe 13-21?

Usare il `for-each` per oggetti di tipo `ArrayInteri`

- Come possiamo modificare la classe `ArrayInteri` in modo tale che sia possibile scrivere il codice seguente?

```
ArrayInteri arrayInt=new ArrayInteri(3,10)

for (int el1: a){
    System.out.print(el1 + ":");

    for (int el2: a) System.out.print(el2 + " ");

    System.out.print("\n");
}
```

- Dobbiamo dichiarare `implements Iterable<Integer>` e quindi definire il metodo:

```
public Iterator<Integer> iterator() {
    return new Iteratore(); //oppure new IteratoreRev();
}
```

- Solo uno dei due iterarori può essere usata per implementare il `for-each`. È quella il cui oggetto è ritornato dal metodo `iterator()` che implementa l'interfaccia `Iterable<Integer>`.

Corso: Paradigmi di Programmazione

Paola Giannini

Classi Anonime e Lambdas

Le classi anonime

Una **classe anonima** è una classe che viene dichiarata all'interno di un'espressione **new**, nel momento in cui viene istanziata:

EspresioneCreaIstanza ::= new Id CorpoClasse | ...

CorpoClasse dichiara le componenti della nuova classe, che sarà:

- una sottoclasse di *Id* se *Id* è una classe
- una classe che implementa *Id* se *Id* è un'interfaccia

La classe anonima, come dice il nome, non ha nome e quindi non può essere usata per istanziare altri oggetti.

Esempio SENZA classi anonime

```
public interface Comparator<T> {  
    public int compare(T a, T b);  
}  
  
public class Persona implements Comparable<Persona>{  
    private String nome;  
    private String cognome;  
    private int eta;  
}  
  
public class ComparaPerNome  
    implements Comparator<Persona> {  
    public int compare(Persona a, Persona b) {  
        return  
            ((a.getNome()).compareTo(b.getNome()));  
    }  
}  
  
public class ComparaPerEta  
    implements Comparator<Persona> {  
    public int compare(Persona a, Persona b) {  
        return  
            ((a.getEta()).compareTo(b.getEta()));  
    }  
}  
  
public static void main(String[] args) {  
    // Creazione ArrayList di Persona  
    ArrayList<Persona> persone = new ArrayList<Persona> .....  
  
    // Ordina per nome  
    Collections.sort(persone, new ComparaPerNome());  
    // Ordina per eta  
    Collections.sort(persone, new ComparaPerEta());  
  
    ....  
}
```

- Per ottenere oggetti di tipo Comparator devo definire classi che implementano l'interfaccia e
- poi fare la **new** di un oggetto quelle classi!

Esempio CON classi anonime

```
public interface Comparator<T> {  
    public int compare(T a, T b);  
}
```

```
public class Persona implements Comparable<Persona>{  
    private String nome;  
    private String cognome;  
    private int eta;  
    ...  
}
```

```
public static void main(String[] args) {  
    // Creazione ArrayList di Persona  
    ArrayList<Persona> persone = new ArrayList<Persona>();  
    // Inserimento elementi nella lista  
    . . .  
    // Ordina per nome  
    Comparator<Persona> cp =new Comparator<Persona>() {  
        public int compare(Persona a, Persona b) {  
            return ((a.getNome()).compareTo(b.getNome()));  
        }  
    };  
    Collections.sort(persone, cp);  
    // Ordina per eta  
    Collections.sort(persone, new Comparator<Persona>() {  
        public int compare(Persona a, Persona b) {  
            return ((a.getEta()).compareTo(b.getEta()));  
        }  
    });  
    . . .  
}
```

- Posso evitare di definire le classi esplicitamente usando le **classi anonime**
- si specifica solo il body e **si devono implementare i metodi astratti dell'interfaccia**

Le classi anonime: uso

- Molte volte una classe è definita per passare un blocco di codice come parametro a un metodo (o a un costruttore). Nell'esempio se usiamo i confronti per nome e/o per eta' solo una volta non occorre definire una classe che implementi Comparator, ma definiamo solo il suo metodo astratto.
- Vedremo classi di questo genere sia nei Thread che nelle interfacce grafiche, ad es: Button, prendono come input il blocco di codice che deve essere eseguito quando avviene l'evento, ad es. il bottone viene premuto.

Lambda-espressioni in Java 8 e interfacce funzionali

- Un'interfaccia che ha **un solo metodo astratto** si chiama **interfaccia funzionale**
- Un modo più sintetico di implementare una interfaccia funzionale, come Hello, introdotto in Java 8, è usare una **lambda-espressione**.
- In Java, una lambda espressione fornisce un modo per creare una **funzione anonima**, introducendo di fatto un nuovo tipo Java.
- In poche parole, si tratta di un **metodo senza una dichiarazione**, una sorta di scorciatoia che consente di scrivere un metodo nello stesso posto in cui serve.

Implementazione interfacce con lambda-espressioni

```
public interface Comparator<T> {  
    public int compare(T a, T b);  
}
```

```
public class Persona implements Comparable<Persona>{  
    private String nome;  
    private String cognome;  
    private int eta;  
    ...  
}
```

```
public static void main(String[] args) {  
    // Creazione ArrayList di Persona  
    ArrayList<Persona> persone = new ArrayList<Persona>();  
    // Inserimento elementi nella lista  
    . . .  
    // Ordina per nome  
    Collections.sort(persone,(a, b)-> ((a.getNome()).compareTo(b.getNome())));  
    // Ordina per eta  
    Collections.sort(persone,(a, b)-> ((a.getEta()).compareTo(b.getEta())));  
    . . .  
}
```

- Possiamo costruire un oggetto di tipo Hello fornendo una **lambda** che
 - prende due parametri e
 - specifica l'espressione che produce il risultato
- cioè **implementa l'interfaccia funzionale Comparator**
- Notate che NON ho specificato
 - il tipo dei parametri (il compilatore li inferisce!) e
 - non ho usato **return** (perchè il body della lambda è una espressione.)

Notazione lambda

In generale la notazione è la seguente

(Lista degli argomenti) -> Espressione

oppure

(Lista degli argomenti)->{ istruzioni; }

Ad esempio

- (1) `(int x, int y) -> x + y`
- (2) `s -> s.length()`
- (3) `() -> 50`
- (4) `(String s) -> { System.out.println("Benvenuto " + s);}`
- (5) `(String s) -> System.out.println("Benvenuto " + s)`

- (1) espressione che prende due interi e restituisce la somma
- (2) espressione che prende una stringa e restituisce la sua lunghezza
- (3) espressione senza argomenti che restituisce il valore 50
- (4) (5) espressione che prende una stringa e non restituisce nulla ma stampa a console

Nota: in un'espressione lambda è possibile **omettere**

- il tipo dei parametri (se sono inferibili dal contesto)
- le parentesi se c'è solo un parametro
- le parentesi graffe del blocco se c'è un solo comando

Corso: Paradigmi di Programmazione

Paola Giannini

Trattamento delle Eccezioni

Situazioni anomale e loro gestione

Durante l'esecuzione di un'applicazione possono sempre verificarsi delle situazioni anomale

- formato errato dei dati in ingresso
- errori di programmazione ad esempio
 - distrazioni o controlli mancanti
 - errori logici nella struttura del programma

Un programma ben scritto dovrebbe prevenire, per quanto possibile, tali eventualità. Come limitare i problemi?

- inserire opportuni controlli per individuare le anomalie prima che l'errore causi la terminazione del programma
- segnalare il problema
- ripristinare una situazione corretta prima di proseguire

Però alle volte non è possibile sapere come recuperare da un errore per cui si può SEGNALARE in un modo più strutturato rispetto al ritornare un valore valore particolare che deve essere interpretato come errore.

Il meccanismo delle eccezioni (1)

- Il meccanismo delle eccezioni di Java è elegante, flessibile, potente, programmabile e perfettamente integrato con il paradigma degli oggetti
 - ad ogni tipo di anomalia associa **una classe di eccezioni**
 - una nuova istanza viene creata quando si verifica il problema
 - la classe ne delinea la natura e le caratteristiche
 - le eccezioni sono organizzate in una **gerarchia estendibile**
- Quando si verifica una situazione anomala:
 - viene **creata un'eccezione** che descrive il problema
 - **il metodo attivo termina** in maniera anomala
 - in contrapposizione alla terminazione corretta con eventuale restituzione di un valore col comando **return**
 - si dice che l'**eccezione viene lanciata**

Il meccanismo delle eccezioni (2)

- I linguaggio Java offre le primitive per
 - **definire** nuove classi di eccezioni
 - **creare** un'istanza di un'eccezione
 - **lanciare** un'eccezione (ovvero segnalare un problema)
 - **catturare** un'eccezione (ovvero farsi carico di risolverla scrivendo una porzione di codice apposita)
 - **propagare** un'eccezione al chiamante (ovvero dichiarare di non essere in grado di gestire certe situazioni anomale)
- A garanzia della robustezza del codice, il compilatore Java controlla che certe eccezioni vengano effettivamente trattate nel codice

Accesso a posizioni inesistenti di array

```
1 public class EsEccezioniOutOfBounds {
2     public static void main(String[] args) {
3         eccArOutOfBounds();
4     }
5     public static void eccArOutOfBounds() {
6         String[] parole = Input.readSeq("> ");
7         int tot = 0;
8         // Controllo sbagliato
9         for (int i=0; i<=parole.length;i++) {
10             tot += parole[i].length();
11         }
12     }
13 }
```

Eseguendo il programma otteniamo il seguente risultato

```
> Pippo
> Pluto
> Paperino
>
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
at codice.EsEccezioniOutOfBounds.eccArOutOfBounds(EsEccezioniOutOfBounds.java:10)
at codice.EsEccezioniOutOfBounds.main(EsEccezioniOutOfBounds.java:3)
```

Accesso a posizioni inesistenti di stringa

```
1 public class EsEccezioniOutOfBounds {
2     public static void main(String[] args) {
3         eccStrOutOfBounds();
4     }
5     public static void eccStrOutOfBounds() {
6         String parola = Input.readString("Parola: ");
7         int pos = Input.readInt("Posizione: ");
8         // Non c'e' il controllo prima di charAt
9         System.out.println("Il carattere e' " + parola.charAt(pos));
10    }
11 }
```

Eseguendo il programma otteniamo il seguente risultato

```
Parola: Pippo
Posizione: 8
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
        String index out of range: 8
        at java.lang.String.charAt(String.java:658)
        at codice.EsEccezioniOutOfBounds.eccStrOutOfBounds(EsEccezioniOutOfBounds.java:9)
        at codice.EsEccezioniOutOfBounds.main(EsEccezioniOutOfBounds.java:3)
```

Errori aritmetici

```
1 public class EsEccezioniOutOfBounds {
2     public static void main(String[] args) {
3         eccDivPerZero();
4     }
5     public static void eccDivPerZero() {
6         int x=Input.readInt("Numeratore: ");
7         int y=Input.readInt("Denominatore: ");
8         System.out.println(x+"/"+y+" = "+ (x/y)+" con resto "+(x%y));
9     }
10}
```

Eseguendo il programma otteniamo il seguente risultato

```
Numeratore: 3
Denominatore: 0
Exception in thread "main" java.lang.ArithmaticException: / by zero
        at codice.EsEccezioniOutOfBounds.eccDivPerZero(EsEccezioniOutOfBounds.java:8)
        at codice.EsEccezioniOutOfBounds.main(EsEccezioniOutOfBounds.java:3)
```

La gerarchia delle eccezioni (1)

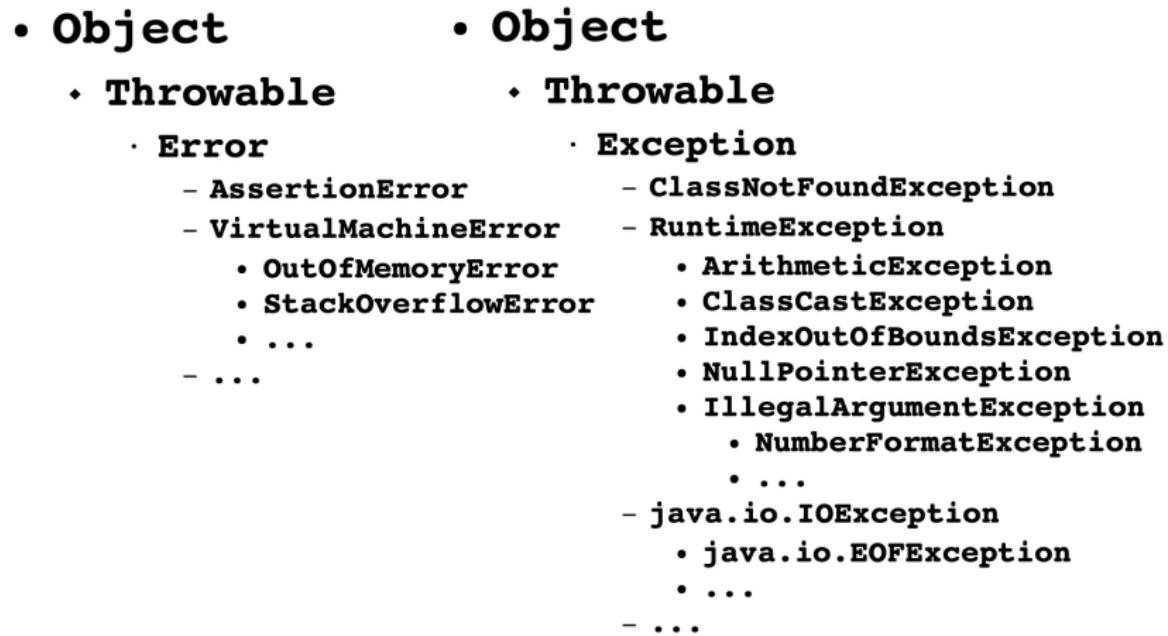
- Le classi di eccezioni sono annidate secondo il meccanismo di ereditarietà per formare la cosiddetta **gerarchia delle eccezioni**
 - molte eccezioni hanno caratteristiche comuni
 - alcune eccezioni sono ovvie specializzazioni di altre
 - es: **accesso a posizioni inesistenti**
- **IndexOutOfBoundsException**
 - **ArrayIndexOutOfBoundsException**
 - **StringIndexOutOfBoundsException**

IndexOutOfBoundsException è **sopraclassificata** di entrambe le classi:
ArrayIndexOutOfBoundsException e
StringIndexOutOfBoundsException.

La gerarchia delle eccezioni (2)

- Tutte le classi di eccezioni sono sottoclassi di Throwable
 - le loro istanze possono essere lanciate (con **throw**)
- La classe Throwable ha due sottoclassi dirette:
 - Error
 - raccoglie eccezioni che indicano errori fatali, sia software che hardware, che difficilmente possono essere prevenuti e gestiti
 - Exception
 - la classe Exception ha, tra le altre, una sottoclasse particolarmente significativa, chiamata RuntimeException che raccoglie tutte le eccezioni più comuni, facilmente prevenibili inserendo maggiori controlli nel codice

Un frammento della gerarchia



Creare le eccezioni

- Ogni oggetto di tipo **Throwable**

- contiene una rappresentazione della pila di ambienti al momento in cui l'istanza dell'eccezione è stata creata
- può contenere un testo di descrizione
- contiene una **causa** che, se diversa da **null**, è il riferimento a un'altra eccezione
 - è quella ritenuta responsabile della presente anomalia.
 - meccanismo delle cosiddette **eccezioni a catena**

- Ogni classe di eccezioni dovrebbe avere almeno 2 costruttori

- uno senza argomenti
- uno con un argomento di tipo **String**

Lanciare le eccezioni

- Negli esempi visti le eccezioni erano lanciate dalla JVM
- Può essere lo sviluppatore di un certo metodo o di una porzione di codice a voler lanciare un'eccezione
 - parametri o dati inattesi e non elaborabili
 - problemi di connessione o di I/O
- Intuitivamente, in queste situazioni si vuole
 - comunicare al chiamante che non è stato possibile svolgere correttamente l'operazione
 - segnalare con precisione la causa del problema
- Java mette a disposizione il comando **throw**

Il comando **throw**: un esempio

- Il tipico schema d'uso del comando **throw**:
 - si testa una certa condizione
 - se l'esito indica un problema si usa il comando **throw**
 - l'espressione crea un oggetto eccezione con **new**

```
public double distanza (Punto other) {  
    if (other==null) {  
        throw new IllegalArgumentException("punto null");  
    }  
    else {  
        double dx=this.x-other.x;  
        double dy=this.y-other.y;  
        return Math.sqrt(dx*dx+dy*dy);  
    }  
}
```

Propagazione di default

- Per default, il metodo chiamante che riceve un'eccezione dal metodo invocato
 - **termina in maniera anomala**
 - **passa l'eccezione** al suo metodo chiamante
- Questa propagazione svuota la pila di ambienti
 - tutti i metodi presenti terminano in modo anomalo
- L'intero **programma termina in maniera anomala stampando**
 - il tipo dell'eccezione
 - il testo di descrizione
 - l'elenco dei metodi che hanno (ri)lanciato l'eccezione

Messaggio di errore: un esempio

- Il messaggio di errore è abbastanza dettagliato da permettere una facile localizzazione del problema
 - l'elenco dei **metodi attraversati dall'eccezione** fornisce una sorta di "fotografia" della pila di ambienti al momento in cui l'anomalia si è verificata

Rivediamo l'esempio dell'eccezione di formato numerico

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "Tre"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
        at jbook.util.Input.readInt(Input.java:85)
        at jbook.util.Input.readInt(Input.java:91)
        at codice.EsEccezioniOutOfBounds.eccFormatoNum(EsEccezioniOutOfBounds.java:9)
        at codice.EsEccezioniOutOfBounds.main(EsEccezioniOutOfBounds.java:3)
```

Il comando `throw`

Comando ::= *CmdTrow* | ...

CmdTrow ::= **throw** *Espressione*

- *Espressione* deve essere di tipo `Throwable`
- Il comando **throw** per prima cosa valuta l'espressione
 - se la valutazione comporta un `errore exc` di qualche tipo allora si termina in maniera anomala con `causa exc`
 - se invece produce un `valore val` non nullo allora si termina in maniera anomala con `causa val`
 - se invece `produce null` allora si crea un'istanza `npt` della classe `NullPointerException` e si termina in maniera anomala con `causa npt`
- Quindi il comando **throw** termina sempre in maniera anomala

Catturare le eccezioni

- Un metodo chiamante può intercettare certe eccezioni (in modo selettivo)
 - per eseguire alcuni comandi
 - per rilasciare certe risorse
 - per impedire la propagazione
 - per evitare la terminazione disastrosa del programma

Comando ::= *CmdTry* | ...

CmdTry ::= **try** Blocco *ClausolaCatch*⁺
try Blocco *ClausolaCatch** **finally** Blocco

ClausolaCatch ::= **catch** (*Param*) Blocco

Un esempio di try-catch-finally

Supponiamo sia definito il metodo seguente:

```
1 public static int ttyDiv(int n) {  
2     try {  
3         int d = Input.readInt("denominatore: ");  
4         return n/d;  
5     } catch (NumberFormatException e) {  
6         System.out.println("volevo un intero: " + e);  
7         return 0;  
8     } catch (ArithmaticNumberException e) {  
9         System.out.println("impossibile dividere per 0");  
10        return 0;  
11    } finally {  
12        System.out.println("Grazie, bye");  
13    }  
14 }
```

Abbiamo

- un **blocco try** alle linee 3 e 4
- due **blocchi catch**: linee 5 e 6 e linee 7 e 8
- un **blocco finally** linea 10

Se il blocco **finally** è assente deve esserci almeno un blocco **catch**, altrimenti possono non essercene.

Il comando **try-catch-finally**

Si esegue il **blocco try**

- in assenza di fallimenti si esegue il **blocco finally**¹
- se lancia l'eccezione ecz si cerca il **primo blocco catch** con ecz compatibile verso il tipo del suo parametro formale
 - se tale blocco esiste l'**eccezione è catturata con successo**:
 - si assegna l'oggetto ecz al parametro formale di quel blocco
 - si eseguono le istruzioni del blocco e poi quelle del blocco **finally**¹
 - se nessun blocco cattura ecz
 - si esegue il blocco **finally**¹
 - poi si termina in maniera anomala, lanciando l'eccezione ecz

¹se presente

Alcune note sui blocchi **catch** e **finally**

- Ogni clausola **catch**
 - deve avere esattamente un parametro formale
 - il parametro deve avere tipo **Throwable** (o sua sottoclasse)
- Il blocco **finally**, se presente, viene eseguito sempre
 - anche in presenza di un comando **return** nel blocco **try** o nel blocco **catch** che ha catturato con successo l'eccezione
- Si raccomanda di inserire nel blocco **finally** le istruzioni
 - per chiudere dei files
 - per rilasciare delle risorse
- I costrutti **try** possono essere annidati tra loro e con altri blocchi

Eccezioni controllate e non

Le eccezioni sono suddivise in due categorie

① eccezioni non controllate (unchecked)

- eventi fatali che non avrebbe senso gestire
- errori logici tra i più comuni, che si verificano solo in codice poco robusto, ma scongiurabili a priori con semplici controlli
- non devono necessariamente essere gestite (**dovrebbero essere evitate con controlli opportuni**)

② eccezioni controllate (checked)

- eventi esterni non evitabili a priori con controlli accurati
 - es: i problemi legati all'accesso al file system in lettura e scrittura
- **devono essere gestite esplicitamente dal programma**
- il compilatore controlla che siano trattate

Eccezioni controllate e non (dichiarazione)

- La distinzione tra le eccezioni controllate e quelle non controllate è guidata dalla gerarchia delle eccezioni
 - sono **non controllate** tutte le eccezioni che estendono `Error` e `RuntimeException`
 - tutte le altre eccezioni sono **controllate**

```
public class ExcUnchecked extends RuntimeException{
    public ExcUnchecked(String msg){
        super(msg);
    }
    public ExcUnchecked(){
        super("ExcUnchecked");
    }
}
public class ExcChecked extends Exception{
    public ExcChecked(String msg){
        super(msg);
    }
    public ExcChecked(){
        super("ExcChecked");
    }
}
```

- Eccezione1 è non controllata
- Eccezione2 è controllata

Catturare o dire che si lanciano le eccezioni checked

- Il codice che chiama i metodi che possono lanciare un'eccezione controllata devono
 - gestire l'eccezione con un blocco **try-catch** o
 - dire esplicitamente, tramite la clausola **throws** che potrebbero lanciare l'eccezione

```
public class LancioEccezioni {  
    public void lancioEccezione1() {  
        throw new Eccezione1("ECCEZIONE 1");  
    }  
    public void lancioEccezione2() throws Eccezione2 {  
        throw new Eccezione2("ECCEZIONE 2");  
    }  
    public void catturoEccezione2() {  
        try {  
            lancioEccezione2();  
        } catch (Exception e) {}  
    }  
    public void nonCatturoEccezione2() throws Eccezione2 {  
        lancioEccezione2();  
    }  
}
```

- Eccezione1 è non controllata
- Eccezione2 è controllata

La clausola **throws**

- La clausola **throws** serve per informare il compilatore che durante l'esecuzione di un certo metodo (o costruttore) possono essere generate certe eccezioni (controllate)
 - viene inserita nella dichiarazione del metodo, subito prima del corpo del metodo
 - i tipi menzionati devono avere **Throwable** come superclasse. La gestione di quelle eccezioni è delegata al chiamante

DichCostruttore ::= *Modificatore** *ParamsDiTipo*? *Id* (*Params*) }
 Lancia? *CorpoCostruttore*

DichMetodo ::= *Modificatore** *ParamsDiTipo*? *Tipo* *Id* (*Params*) }
 Lancia? *CorpoCostruttore*

Lancia ::= **throws** *IdQ* (, *IdQ*)*

Definire le proprie eccezioni

- Potete usare le eccezioni predefinite di Java quando è possibile
 - cercando ogni volta quella più adatta alle proprie esigenze
- Se ritenuto necessario o più conveniente, definire nuove classi di eccezioni in questo caso ha più senso definire **eccezioni checked** e
 - per convenzione definite come **sottoclassi di Exception**

Testare che vengano sollevate le eccezioni (1)

- Vogliamo che un test abbia successo se sono sollevate delle eccezioni (nella chiamata dei metodi)
- Questo ci pone un problema dovuto al fatto che se un'eccezione viene sollevata NON si esegue l'istruzione successiva a quella che l'ha sollevata.
- Vediamo un esempio. Nella classe di test si definiscono delle asserzioni, che specificano che un metodo DEVE sollevare un'eccezione (se non la solleva FALLISCE)
- Si può specificare il tipo dell'eccezione che deve essere sollevata e anche il messaggio che deve contenere.

Testare che vengano sollevate le eccezioni (2)

- Un metodo, NON CONSIGLIATO, è fare uso nei test del costrutto **try-catch** per assicurarsi che l'eccezione venga sollevata e
 - fallire se non viene sollevata (cioè si esegue l'istruzione seguente a quella che la doveva sollevare)
- Il secondo metodo che è possibile con JUnit5 è attraverso asserzioni che hanno successo/falliscono se vengono sollevate eccezioni (di classi specifiche) e che ritornano anche l'oggetto eccezione lanciato.

Metodo che testa il sollevarsi o meno con try-catch NON consigliato!

```
@BeforeEach
public void init() {
    ck = new CheckingAccount(30);
}
@Test
public void testWithdrawExceptionTryCatch() {
    try {
        ck.deposita(200);
        ck.preleva(201);
        fail("Aspettavo un'eccezione");
    } catch (FondiInsuffException exc) {
        assertEquals(1, exc.getAmount(), 0.0001);
        assertEquals("Fondi non sufficienti", exc.getMessage());
    }
    try {
        ck.deposita(200);
        ck.preleva(450);
        fail("Aspettavo un'eccezione");
    } catch (FondiInsuffException exc) {
        assertEquals(50, exc.getAmount(), 0.0001);
        assertEquals("Fondi non sufficienti", exc.getMessage());
    }
}
```

- DOVETE usare questo pattern per assicurarvi che il metodo **fallisce se non viene generata l'eccezione** e **ha successo se viene sollevata l'eccezione voluta**
- SVANTAGGI
 - è necessario capire il funzionamento del try-catch.
 - il codice del test è complicato

Metodo che testa il sollevarsi o meno di MiaEcccezione con JUnit5

```
@Test
public void testWithdrawExceptionAssThr() {
    FondiInsuffException exc = Assertions.assertThrows(FondiInsuffException.class, () ->
        ck.deposita(200);
        ck.preleva(201);
    );
    assertEquals("Fondi non sufficienti", exc.getMessage());
    assertEquals(1, exc.getAmount(), 0);
    FondiInsuffException exc1 = Assertions.assertThrows(FondiInsuffException.class, () ->
        ck.deposita(200);
        ck.preleva(450);
    );
    assertEquals("Fondi non sufficienti", exc1.getMessage());
    assertEquals(50, exc1.getAmount(), 0);
}
```

- Si può testare il sollevarsi di più di una eccezione.
- Il codice è semplice.
- Esiste anche una assert che dice che non vengono sollevate eccezioni.

```
assertDoesNotThrow(() -> /*Codice che non causa l'eccezione*/)
```

assertThrows di JUnit5

Il metodo `assertThrows ()` della classe `Assertions` prende i seguenti **parametri**:

- ① Un oggetto di una classe che specifica il tipo dell'eccezione sollevata.
- ② Una **lambda espressione** con zero argomenti che è il codice da testare, che può essere:
 - una espressione da valutare o
 - sequenza di istruzioni da eseguire
- ③ Un messaggio di errore opzionale. (Se c'è è ritornato quando l'eccezione non è sollevata!)

Il metodo `assertThrows ()` **restituisce** l'oggetto eccezione generata che quindi può essere usato per testare il messaggio contenuto.

Corso: Paradigmi di Programmazione

Paola Giannini

Cenno a Espressioni Regolari in Java

Espressioni regolari

- Espressioni regolari (ER) sono molto utili per
 - controllare,
 - dividere,
 - trasformare stringhe
- I compilatori usano ER come primo passo per tokenizzare il codice, rimuovere bianchi, commenti.
- Combinazioni di **operatori** e **letterali** (simboli terminali) per creare dei **pattern**
 - i **letterali** sono i caratteri.
 - L'espressione regolare carattere a denota la stringa "a", 1 denota "1" e così via per la maggior parte dei caratteri

Gli operatori

Alcuni caratteri, **operatori** hanno un significato speciale. Gli operatori principali sono:

- | : alternanza $a|b$ significa a oppure b
- "" : concatenazione ab significa a seguito b
- * : ripetizione $a*$ significa zero o più a ("", a , aa, \dots)
- + : ripetizione a^+ significa una o più a (a , aa, \dots)
- ? : facoltatività $a?$ significa "" oppure a
- () : raggruppamento (per dare scope ad un operatore; esempio $(ab)^*$ significa ab ripetuti zero o più volte)

Le ER in Java (come in altri linguaggi) sono rappresentate da stringhe

Inoltre si usa una sintassi molto compatta, in particolare per quanto riguarda l'identificazione di caratteri:

- [abc] identifica uno fra a, b o c sarebbe (a | b | c)
- [^abc] tutti i caratteri eccetto a, b
- [0-9] : range da 0 a 9 (una cifra)
- [a-z[A-Z]] : unione (si possono omettere le quadre interne)
- [a-z&&[^bc]] intersezione

Quali stringhe sono generate/matchate da:

- ab*a
- [a-z[A-Z]] [[0-9] [a-z] [A-Z]]*

Per identificare caratteri frequenti ci sono alcune classi predefinite:

- . (il punto) qualsiasi carattere (**wild card**)
- \d è equivalente a [0-9] mentre \D è equivalente a [^0-9]
- \s uno spazio, \t tab \n a capo, \S non \s (non uno spazio)
- \w un carattere (da identificatore) è equivalente a [a-zA-Z_0-9]
(lettere maiuscole o minuscole, cifre o underscore) \W è equivalente a ^\w

PROBLEMA: i pattern sono stringhe e \ è un carattere che ha una interpretazione nelle stringhe per cui i pattern devono essere scritti con 2 \.
Esempio: \d deve essere scritto: \\d e se vogliamo matchare il carattere punto dobbiamo scrivere \\.

Supporto a ER nella classe String

Metodi di istanza della classe String

- **public boolean** matches(String regex) ritorna **true** se la stringa su cui è invocato matcha il parametro regex
- **public** String[] split(String regex) ritorna un array di stringhe divise dalle sottostringhe che matchano il pattern regex
- **public** String replaceAll(String regex, String replacement) rimpiazza ogni sottostringa della stringa che matcha regex con la sequenza di rimpiazzamento. Es: rimpiazzare aa con b in aaaaaaaa risulta in bbba.
- **public** String replaceFirst(String regex, String rep) rimpiazza solo la prima.
- C'è anche una classe Pattern con metodi molto più sofisticati.

Corso: Paradigmi di Programmazione

Paola Giannini

Multithreading

Single-threading e multi-threading

Consideriamo il flusso di un programma Java:

- inizialmente si esegue il metodo `main(...)` di una classe
- si prosegue quindi sequenzialmente, di istruzione in istruzione
 - costrutti di controllo come **if-else** e **while** alterano il flusso
 - invocazioni di metodo **saltano** ad eseguire il metodo invocato, quindi **ritornano** al chiamante e proseguono
 - le eccezioni possono causare dei salti a clausole **catch** o uscite dal metodo

In ogni caso, esiste sempre **una sola istruzione in esecuzione** ad ogni dato momento

- Il modello di esecuzione con un unico flusso di esecuzione (**thread**) è detto **single-threading**. Un programma con un unico flusso di esecuzione è detto **single-threaded**
- È possibile che un programma abbia **contemporaneamente attivi più flussi di esecuzione**
- Si parla di modello **multi-threading** e di programmi **multi-threaded**

Un **thread** è un **processo light-weighted**: l'unità più piccola di codice gestita dallo scheduler del sistema operativo.

- Ogni thread, una volta avviato, ha
 - una parte di **memoria locale al thread**
 - costituita dallo **stack dei record di attivazione** dei metodi in esecuzione
 - quindi le variabili di tipo primitivo sono locali al thread
 - accesso all'**heap**, ovvero la memoria in cui esistono le istanze di oggetti, **è condiviso fra tutti i thread**
 - quindi thread diversi possono comunicare scambiandosi dati attraverso oggetti condivisi nello heap
- I thread esistono all'interno della macchina virtuale
- I thread **non sono entità del linguaggio**
- Un thread è rappresentato nel linguaggio da un'istanza della classe di sistema **Thread**
 - La classe Thread offre numerosi metodi per gestire i thread
 - Vedremo nel seguito solo i principali si possono consultare le API relative per i dettagli

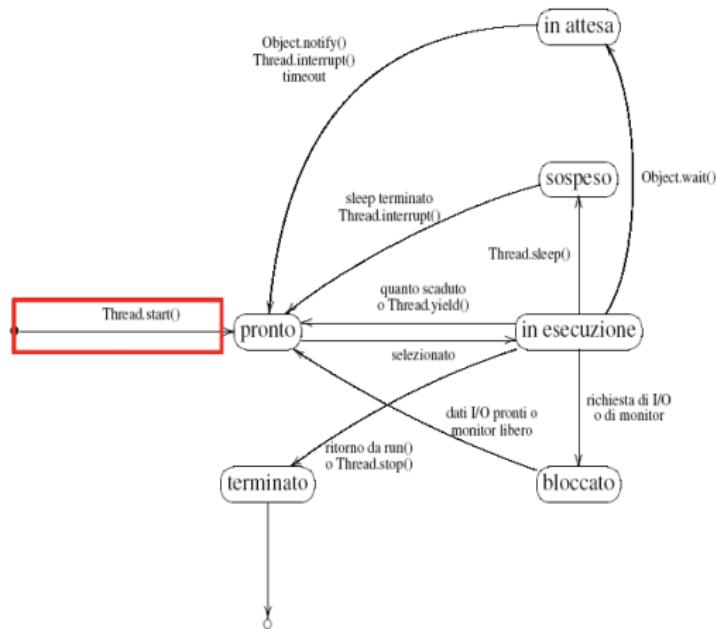
Ciclo di vita di un thread

- Proprio come un programma attraversa le fasi di avvio, esecuzione, terminazione, così i thread hanno un proprio **ciclo di vita**
- All'inizio dell'esecuzione del programma esiste già un thread: quello **principale**
 - si tratta del thread che esegue il metodo `main(...)`
 - in un programma single-threaded, non se ne creano altri

Un thread può poi crearne e avviare altri nel corso della propria esecuzione

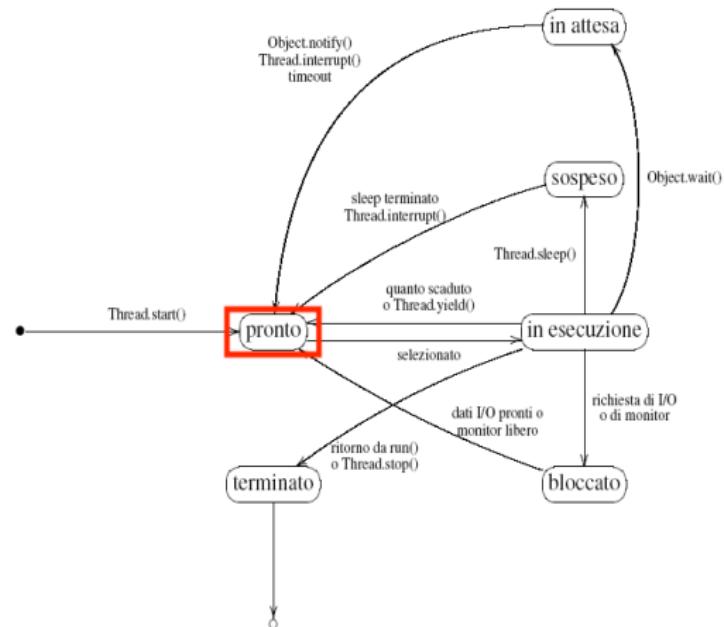
Stati di un thread (1): creazione

- Un Thread viene creato come di consueto con una espressione new
- Inizialmente, il thread non è attivo
- Diventa attivo invocando su di esso il metodo start()



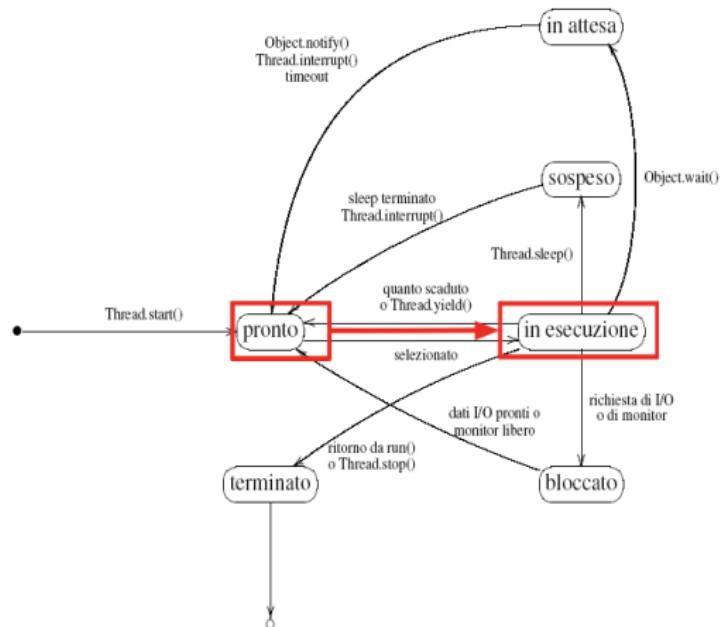
Stati di un thread(2): pronto

- Nello stato di pronto, il thread è pronto per essere eseguito
- Non sempre questo coincide con l'esecuzione
 - Un programma può avere più thread pronti che CPU disponibili per eseguirli
 - Si fa a turno!



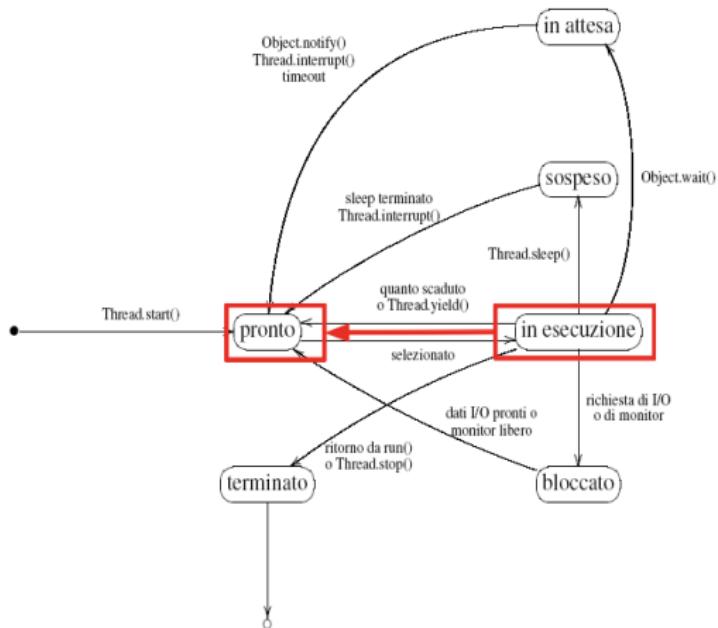
Stati di un thread(3): in esecuzione

- Quando arriva il suo turno, un thread pronto viene selezionato e messo in esecuzione
- In questo stato, il thread esegue le istruzioni secondo il suo programma
 - vedremo dopo come assegnare un programma a un thread



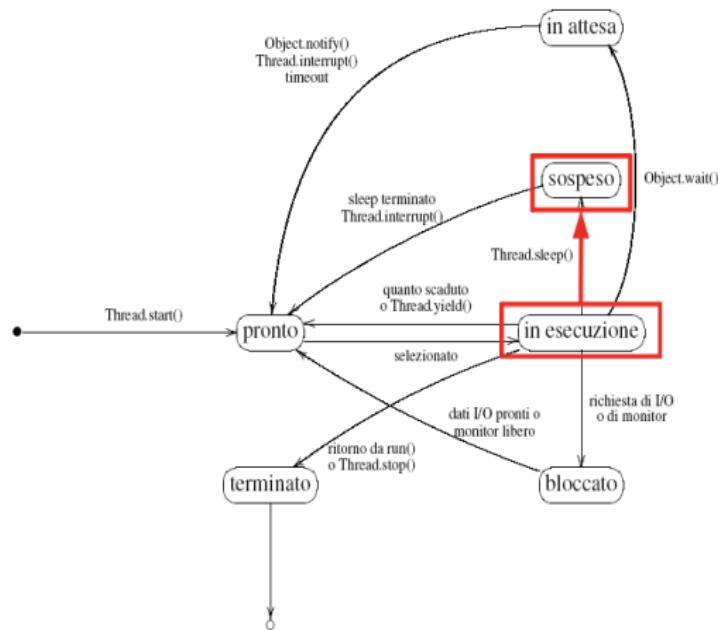
Stati di un thread(4): ritorno a pronto

- Si esce dallo stato di esecuzione se...
 - Scade il **quanto di tempo assegnato** al thread
 - è terminato il **turno** di questo thread
 - il thread torna in pronto
 - un altro thread verrà selezionato e portato in esecuzione



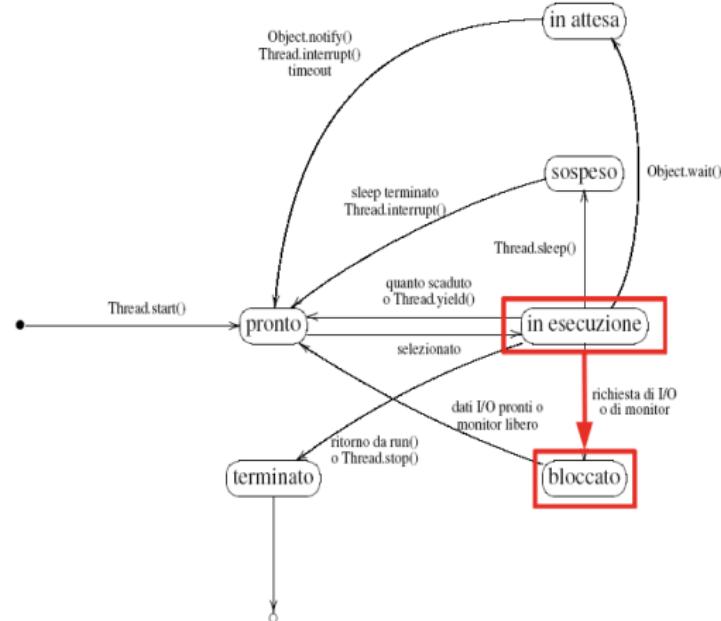
Stati di un thread(5): passaggio a sospeso

- Si esce dallo stato di esecuzione se...
 - Il thread invoca una `sleep()`
 - il thread si addormenta per un certo tempo
 - al risveglio (naturale o forzato da una interruzione), il thread passerà in pronto
 - nel frattempo, un altro thread pronto può andare in esecuzione



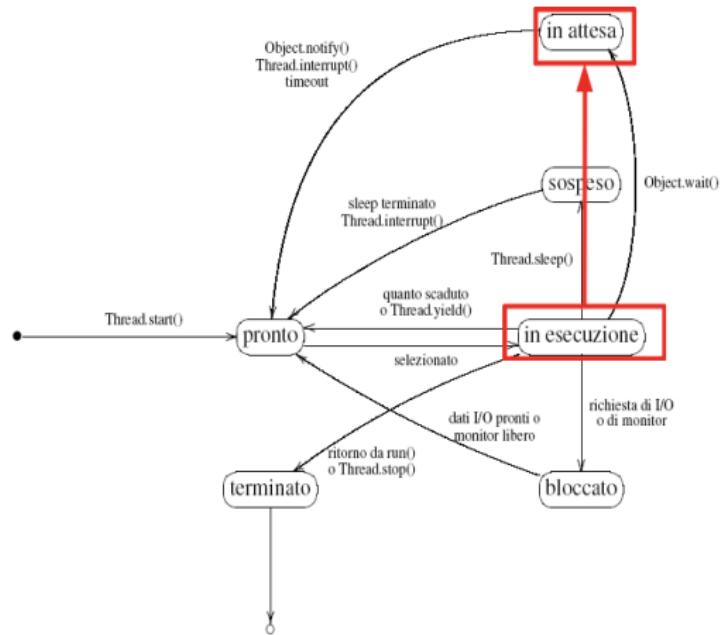
Stati di un thread(6): passaggio a bloccato

- Si esce dallo stato di esecuzione se...
 - Il thread esegue una operazione di I/O
 - il thread viene bloccato finché non ci sono dati
 - al completamento dell'I/O, il thread passerà in pronto
 - nel frattempo, un altro thread pronto può andare in esecuzione
 - comportamento simile anche per i **monitor esplicativi**



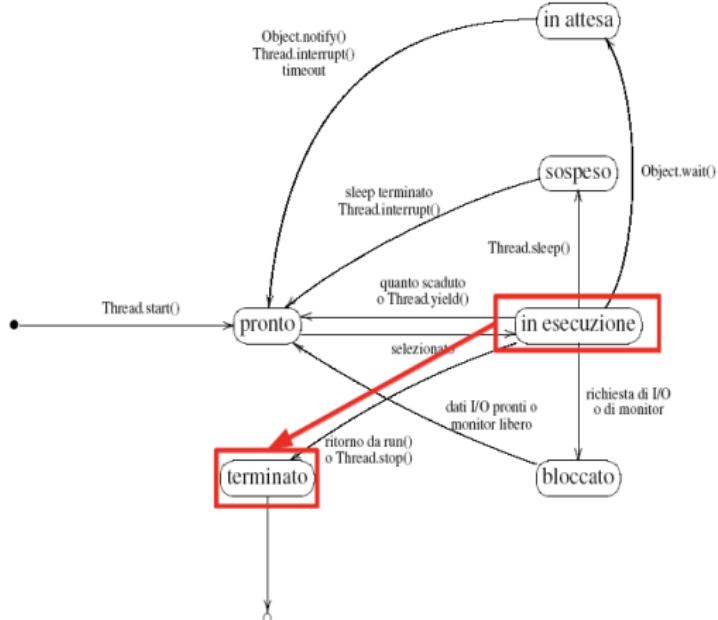
Stati di un thread(7): in attesa

- Si esce dallo stato di esecuzione se...
- Il thread invoca una `wait()`
 - il thread si sospende in attesa che un altro thread invochi una `notify()` sull'oggetto della `wait()`
- al risveglio (naturale o forzato da una interruzione), il thread passerà in pronto
- nel frattempo, un altro thread pronto può andare in esecuzione



Stati di un thread(8): terminato

- Si esce dallo stato di esecuzione se...
 - Il thread termina
 - terminazione **naturale**: il thread completa il proprio programma
 - terminazione **forzata**: il thread viene **ucciso** da un altro thread che invoca su di esso il metodo `stop()`
 - l'oggetto Thread rimane **accessibile** anche se il thread è terminato



Creazione di un thread (1)

- Per creare un thread, servono due entità:

- un Thread, che rappresenta il flusso di esecuzione
- un Runnable, che fornisce il programma da eseguire
- Runnable è un'interfaccia, e Thread la implementa (in modo banale, cioè con un metodo che non fa niente) per cui, è possibile usare un solo oggetto che copra entrambi i ruoli, oppure due oggetti distinti

```
public interface Runnable {  
    public void run();  
}
```

```
public class Thread implements Runnable {  
    public void run(){ /* codice da girare */ };  
}
```

Creazione di un thread (2)

Primo metodo:

- si crea una sottoclasse di Thread, ridefinendone il metodo run()

```
class MioThread extends Thread {  
    public void run(){  
        /* codice da girare */  
    }  
}
```

- si avvia il thread invocandone il metodo start() che fa passare il thread nello stato pronto

```
MioThread mt = new MioThread();  
mt.start();
```

Creazione di un thread (3)

Secondo metodo:

- si definisce una propria classe che implementa Runnable

```
class MioRunnable implements Runnable {  
    public void run(){  
        /* codice da girare */  
    }  
}
```

- si definisce un Thread passando l'oggetto di tipo Runnable al suo costruttore, quindi lo si avvia con start()

```
MioRunnable mr = new MioRunnable();  
Thread t = new Thread(mr);  
t.start();
```

Creazione di un thread (3)

Variazione sul secondo metodo:

- invece di definire una propria classe che implementa `Runnable` se la classe non serve ad altro si può implementare l'interfaccia (che contiene un solo metodo) con una lambda-espressione
- si definisce un `Thread` passando passando la lambda-espressione al suo costruttore, quindi lo si avvia con `start()`

```
Thread t = new Thread(() -> /* codice da girare */  
t.start();
```

Creazione di un thread: differenze nella creazione

- Il primo metodo è più semplice, ma meno flessibile
 - si ha un solo oggetto da gestire
 - l'oggetto **deve essere sottoclassificato** di Thread, quindi non può estendere alcuna altra classe
- Il secondo metodo è più flessibile, ma (poco) più complicato ,
 - qualunque oggetto può implementare Runnable e quindi è possibile usare l'ereditarietà per collocare l'oggetto in una gerarchia, ma
 - bisogna gestire due oggetti distinti (nel caso in cui non si usi una lambda-espressione)
- La scelta migliore dipende dai casi!

Thread sottoclasse di Thread e uno che implementa Runnable

```
public class ElefanteThread extends Thread {  
    public void run() {  
        int i = 2;  
        while (!Thread.interrupted()) {  
            System.out.println(i + " elefanti si dondolavano... in Thread");  
            i++;  
        }  
    }  
}  
  
public class Elefante { /* campi e metodi della classe */  
  
    public class ElefanteRunnable extends Elefante implements Runnable {  
        public void run() {  
            int i = 2;  
            while (!Thread.interrupted()) {  
                System.out.println(i + " elefanti si dondolavano... in Runnable");  
                i++;  
            }  
        }  
    }  
}
```

Vari modi di creazione di un thread

```
1 public class DondolaElefanti {  
2     // Oggetto di tipo ElefanteThread sottoclasse di Thread  
3     Thread tET = new ElefanteThread();  
4     // Oggetto di tipo ElefanteRunnable che implementa Runnable  
5     Thread tER = new Thread(new ElefanteRunnable());  
6     // Oggetto di una classe anonima che implementa Runnable  
7     Thread tEA = new Thread(new Runnable(){  
8         public void run() {  
9             int i = 2;  
10            while (!Thread.interrupted()) {  
11                System.out.println(i + " elefanti si dondolavano...in Classe Anonima");  
12                i++;  
13            }  
14        }  
15    }  
16 );  
17     // Una lambda-espressione che implementa Runnable  
18     Thread tEL = new Thread(() -> {  
19         int i = 2;  
20         while (!Thread.interrupted()) {  
21             System.out.println(i + " elefanti si dondolavano...in Lambda");  
22             i++;  
23         }  
24     });  
25     System.out.println("Premi Invio per terminare");  
26  
27     tET.start(); tER.start(); tEA.start(); tEL.start();  
28     Input.readString();  
29     tEL.interrupt(); tER.interrupt(); tEA.interrupt(); tET.interrupt();  
30 }  
31 }
```

Controllare l'esecuzione

- La classe Thread offre numerosi metodi per controllare l'esecuzione del thread corrispondente
- Alcuni metodi sono **static**, e agiscono sul thread corrente:
 - `sleep(m)` - sospende il thread corrente per `m` millisecondi
 - `yield()` - rilascia volontariamente la CPU e passa nello **stato di pronto**
 - `interrupted()` - restituisce **true** se il thread ha ricevuto una richiesta di interruzione
 - `currentThread()` - restituisce un riferimento al thread corrente

Metodi di istanza: join e metodi informativi

- Il metodo `join` è un metodo di istanza permette al thread corrente di mettersi in pausa per attendere il completamento dell'esecuzione di un altro thread:

```
thr.join()
```

aspetta che `thr` completi l'esecuzione (ci sono overloads in cui si può specificare come per sleep un certo numero di millisecondi).

- Su un thread possono essere invocati numerosi metodi di istanza per ottenere informazioni varie

- `getId()` - restituisce l'id del thread
- `getName()` - restituisce il nome del thread
- `getPriority()` - restituisce la priorità del thread
- `getState()` - restituisce lo stato in cui si trova il thread
- `isAlive()` - restituisce true se il thread è vivo
- `isInterrupted()` - restituisce true se il thread ha ricevuto una richiesta di interruzione
- ecc.....

Implementazione multithreaded del Mergesort

L'array da ordinare

- viene divisa in 2 e copiata in 2 array
- i due sotto-array vengono passati a 2 thread che ne fanno il mergesort
- quando i thread finiscono l'esecuzione
- viene fatto il merge dei risultati

Condizioni di terminazione (1)

- Normalmente, la JVM termina quando termina il programma in esecuzione
- Questa nozione ha senso solo se c'è un unico thread in esecuzione
- Cosa accade quando esistono più thread?
 - il **thread iniziale** potrebbe terminare mentre sono ancora in esecuzione altri thread creati successivamente
 - in alcuni casi, non si vuole terminare: c'è ancora lavoro da fare, meglio aspettare che terminino tutti i thread
 - in altri, si vuole terminare: gli altri thread sono **di servizio** e non hanno senso senza il principale

Condizioni di terminazione (2)

- I thread sono distinti in due classi:
 - **thread demoni**: sono thread di servizio, sono utili soltanto mentre altri thread sono al lavoro esempio: un thread **timer**, o un thread per **garbage collection**
 - **thread non-demoni**: sono quelli che fanno il lavoro vero, quello che rende utile il programma
- La JVM termina quando tutti i **thread non-demoni** sono terminati
 - se in quel momento sono in esecuzione thread demoni, vengono fermati e il programma esce comunque

Comunicazione e interazione

- Raramente i problemi si prestano a una soluzione concorrente in cui i thread sono totalmente indipendenti
- Più frequentemente, i thread devono comunicare, interagire, coordinarsi
- Varie modalità di interazione
 - **interruzione** (agire sul controllo)
 - **condivisione della memoria**
 - **sincronizzazione**
 - **segnalazione e notifica**

Le interruzioni

- Se un thread T1 chiama il metodo `interrupt()` su un thread T2, viene impostato il **flag di interruzione** di T2
- Se al momento della chiamata T2 era sospeso o in attesa, il metodo che ha causato l'interruzione termina lanciando una `InterruptedException`
 - l'eccezione va gestita poi dal codice di T2

NOTA: questa è la ragione per cui la chiamata dei metodi `sleep` e `join` devono essere in un **try-catch**

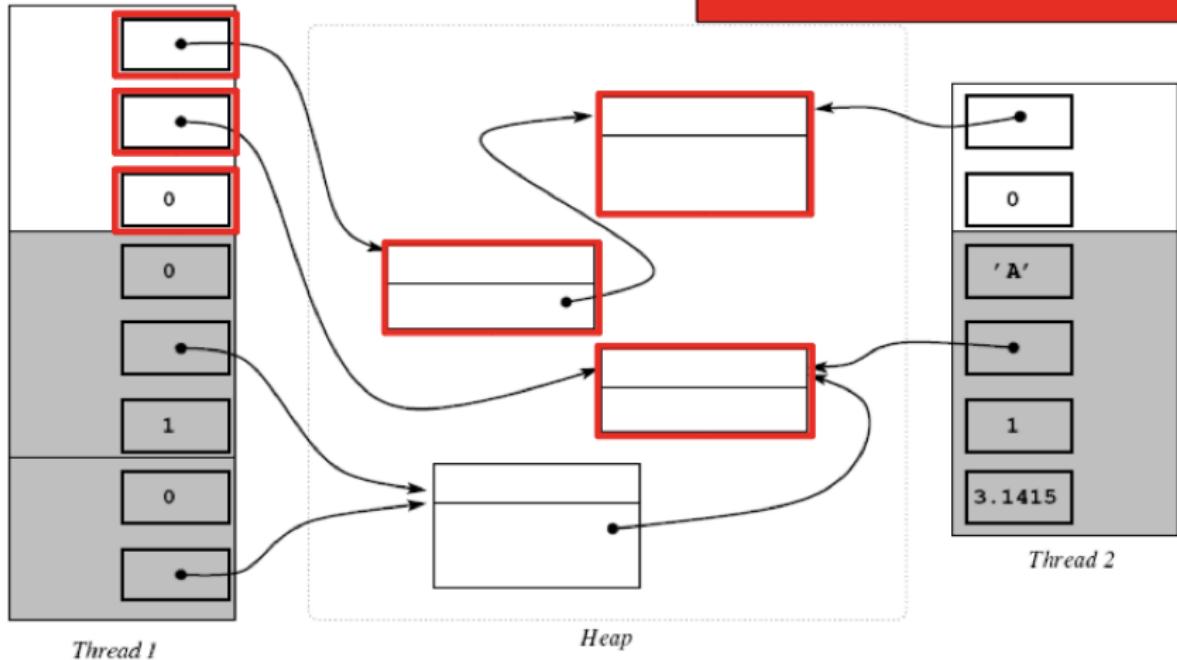
- Negli altri casi, il flag rimane impostato
 - può essere letto e cancellato da T2 chiamando il metodo statico `interrupted()`
 - può essere letto e non cancellato da qualunque thread chiamando il metodo di istanza `isInterrupted()` su T2
- Se T2 entra in sospeso o in attesa con il flag impostato, il metodo che ha causato la sospensione o l'attesa termina immediatamente lanciando una `InterruptedException`
- **Torniamo a vedere l'esempio degli elefanti!**

Condivisione della memoria

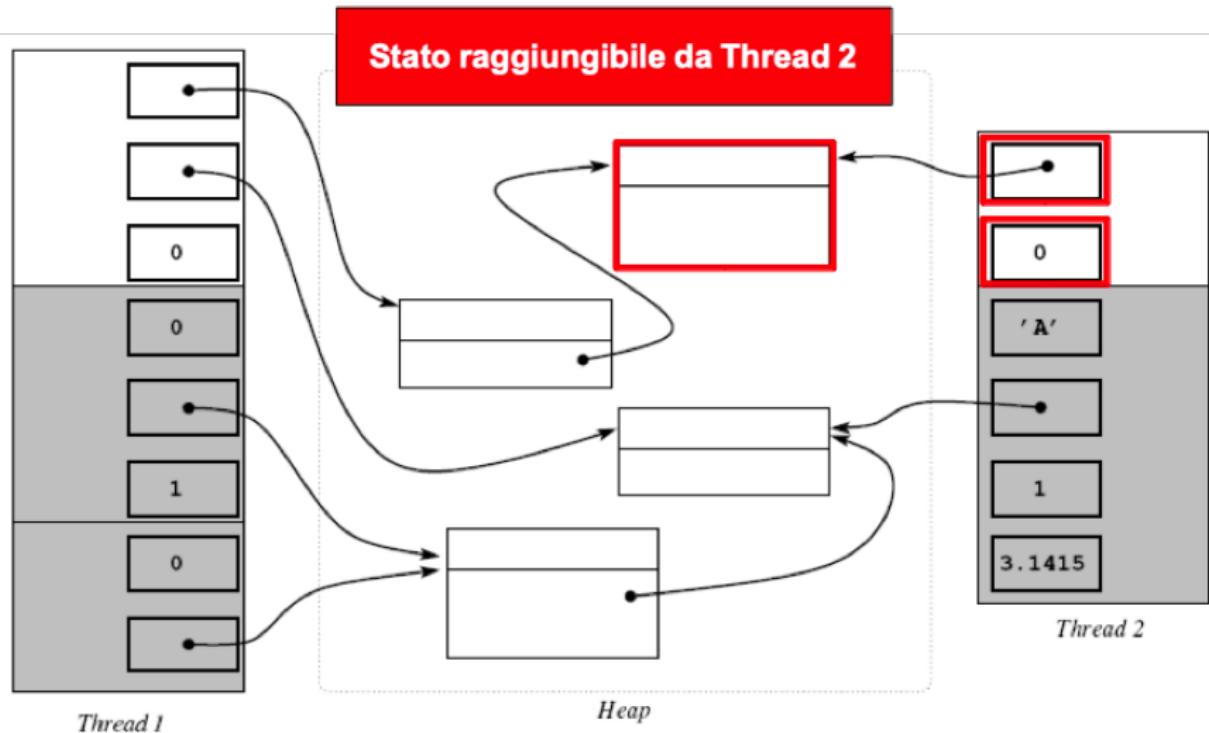
- Tutti i thread condividono lo stesso heap
 - in particolare, tutti gli oggetti allocati con **new**, con le loro variabili di istanza, e tutte le classi, con le loro variabili statiche, sono visibili a tutti i thread
- Ciascun thread ha una propria pila di ambienti
 - la pila del thread iniziale parte con l'ambiente locale del `main(...)`
 - la pila di tutti gli altri thread parte con l'ambiente locale del rispettivo `run()`
- Perché un dato oggetto sia raggiungibile, occorre che lo si possa accedere partendo dall'ambiente locale del thread

Esempio (1)

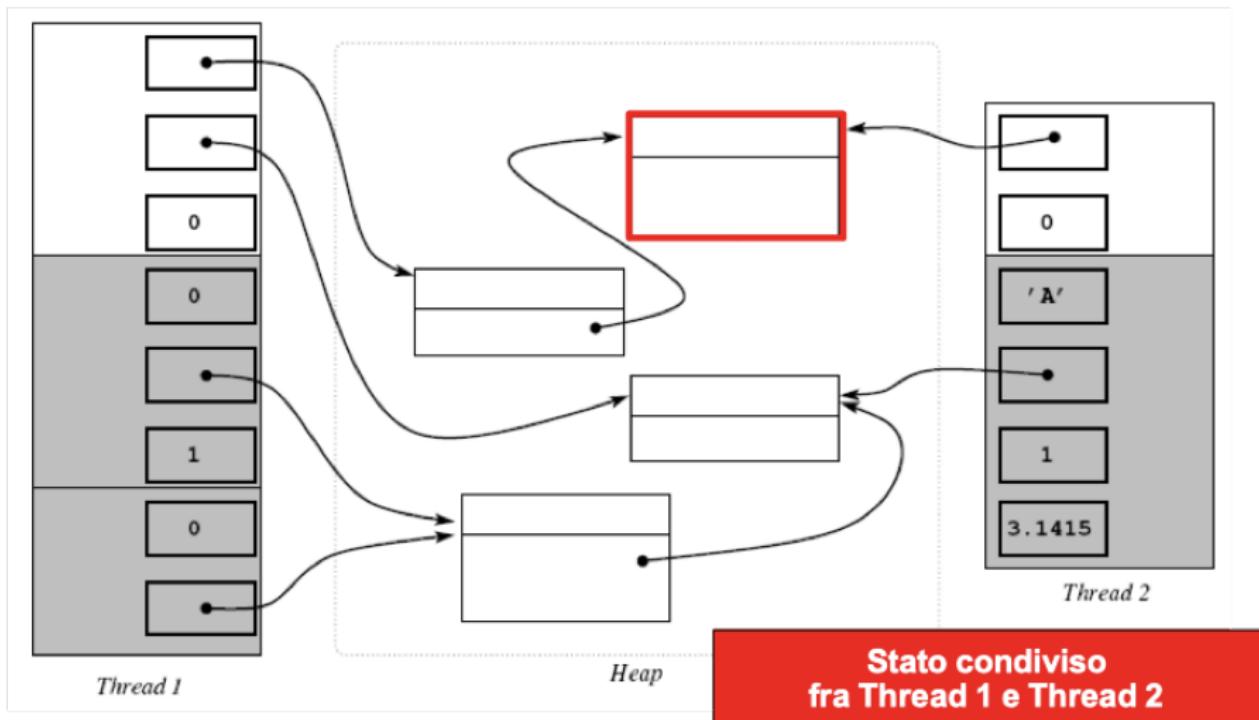
Stato raggiungibile da Thread 1



Esempio (2)



Esempio (3)



Condivisione della memoria

- La presenza di parti condivise dello stato può causare però problemi imprevisti...
- Due thread (o più) possono accedere alla stessa area di memoria.
- Prendiamo un esempio di **Produttore/Consumatore** banale.
- Un Produttore genera un numero da 0 a 9, lo memorizza in un oggetto Contenitore poi lo stampa. Fra la generazione di un numero e l'altro il produttore **dorme per un tempo casuale**.
- Un Consumatore che è **vorace** vuole consumare l'intero nel Contenitore appena disponibile.

Sincronizzazione

- Un modo per prevenire **race conditions** è fare in modo che l'accesso al Contenitore sia **sincronizzato**.
- La sincronizzazione ha 2 aspetti distinti.
 - ❶ da una parte il Produttore ed il Consumatore NON devono accedere al Contenitore simultaneamente.
 - ❷ Il Produttore deve aver un modo per dire al Consumatore che ha prodotto un nuovo valore e viceversa il Consumatore deve avere un modo per dire che ha consumato un valore.

Monitor (1)

- Il costrutto base per regolare l'accesso simultaneo ad un oggetto in Java è il **monitor**
- Un thread può richiedere di prendere un monitor
 - se nessun altro thread possiede il monitor in quel momento, il monitor viene assegnato al thread richiedente e l'elaborazione continua
 - terminato l'uso, il thread rilascerà il monitor
 - **altrimenti**, il thread richiedente viene posto nello stato bloccato
 - quando il thread che possedeva il monitor lo rilascia, tutti i thread **bloccati su quel monitor** tornano nello stato di **pronto**, e tentano di nuovo di acquisire il monitor. Solo uno ci riuscirà, gli altri tornano in **bloccato!**

Monitor (2)

- Proprietà fondamentale:
in ogni istante, al più un thread può possedere un dato monitor
- Alcune note
 - è possibile che un monitor sia libero - nessun thread lo possiede
 - è possibile che lo stesso thread tenti di prendere un monitor che già possiede
 - in questo caso, il tentativo di prendere il monitor ha immediatamente successo
 - quando un thread termina, rilascia tutti i suoi monitor

Monitor e oggetti

- Ogni oggetto di tipo Object ha un proprio monitor associato
 - tutti gli oggetti, che sono sottoclassi di Object, hanno dunque un monitor associato
- Si noti che:
 - le classi hanno un monitor
 - in realtà, è l'istanza di Class associata alla classe - e dunque un oggetto - che lo ha)
 - i valori di tipi primitivi non hanno un monitor
 - ma i tipi involucro sono sottoclassi di Object, e dunque lo hanno

Regioni critiche

- Il Produttore ed il Consumatore NON devono accedere al Contenitore simultaneamente. Cioè non ci devono essere chiamate di `get()` e di `put(..)` contemporanee.
- Per questo si usano metodi **synchronized** (oppure regioni synchronized che vediamo dopo):

```
public class Contenitore {  
  
    public synchronized void put(int i) {  
        .....  
    }  
  
    public synchronized int get() {  
        .....  
    }  
}
```

Fare il lock di un oggetto

- Ad ogni oggetto istanza di Contenitore viene associato un lock che deve essere acquisito per poter eseguire i metodi sincronizzati (put e get).
- Se il thread T1 cerca di eseguire uno dei metodi quando un diverso thread T2 ha il lock, T1 si blocca (finché il monitor non è di nuovo libero).
- I lock sono **rientranti**. Nell'esempio di sotto una volta acquisito il lock per eseguire un metodo, esempio a(), un thread può anche eseguire un altro metodo b().

```
public class Reentrant {  
    public synchronized void a() {  
        b();  
        System.out.println("Sono qui, in a()");  
    }  
    public synchronized void b() {  
        System.out.println("Sono qui, in b()");  
    }  
}
```

Coordinamento (che non funziona)!

```
public class Contenitore {  
    private boolean disponibile=false;  
    private int contenuto;  
  
    public synchronized void put(int i) {  
        if (!disponibile) {  
            disponibile = true;  
            contenuto = i;  
        }  
    }  
  
    public synchronized int get() {  
        if (disponibile) {  
            disponibile = false;  
            return contenuto;  
        }  
        // altro return  
    }  
}
```

- Una `get()` prima che ci sia stata una `put(...)` (`disponibile==false`) viene persa.
- Simile per la `put(...)` prima che il contenuto sia stato prelevato dalla `get()`
- Così **NON funziona!**

Coordinamento: wait/notify

```
public class Contenitore {  
    private boolean disponibile=false;  
    private int contenuto;  
  
    public synchronized void put(int i) {  
        while (disponibile) {  
            try { this.wait(); } catch (InterruptedException e) {}  
        }  
        contenuto = i;  
        disponibile = true;  
        this.notifyAll(); //notifica il Consumatore: valore disponibile  
    }  
  
    public synchronized int get() {  
        while (!disponibile) {  
            try { this.wait();} catch (InterruptedException e) {}  
        }  
        disponibile = false;  
        this.notifyAll(); //notifica il Produttore: valore prelevato  
        return contenuto;  
    }  
}
```

wait() e notify()

- La classe Object implementa tre metodi (di istanza):
 - wait() - sospende il thread chiamante sull'oggetto
 - notify() - sveglia un thread in attesa sull'oggetto
 - notifyAll() - sveglia tutti i thread in attesa sull'oggetto
- Tutti gli oggetti ereditano da Object , e quindi possiedono questi metodi

wait()

- Quando un thread `thr` invoca il metodo `wait()` su un oggetto `obj` (di cui detiene il monitor), `thr` rilascia il monitor di `obj` e passa in attesa, finché:
 - (i) un altro thread invoca `notify()` o `notifyAll()` su `obj`;
 - (ii) un altro thread invoca `interrupt()` su `thr` (nel qual caso la `wait()` lancia una `InterruptedException`);
- Dopo (i), o (ii), `thr` ri-acquisisce il monitor di `obj` (eventualmente competendo con altri thread) e riprende l'esecuzione, ritornando dalla `wait()`

notify() e notifyAll()

- Quando un thread `thr` invoca il **metodo `notify()`** su un oggetto `obj` (di cui detiene il monitor),
 - uno a caso fra i thread in attesa (se ce ne sono) a causa di una loro invocazione alla `wait()` su `obj` viene **svegliato**
 - il thread svegliato tenta di accedere al monitor di `obj` questo può accadere solo dopo che `thr`, che detiene il monitor di `obj`, lo rilascia!
 - una volta ottenuto il monitor, passa nello stato di **pronto**
- In pratica, la `notify()` **sveglia un thread bloccato** dalla `wait()` corrispondente
- Quando un thread `thr` invoca il **metodo `notifyAll()`** su un oggetto `obj` (di cui detiene il monitor, come nel caso di `notify()`),
 - **tutti** i thread in attesa (se ce ne sono) su `obj` vengono svegliati
 - **ciascuno** di loro tenta di ottenere il monitor di `obj` e continuare l'esecuzione come per la `notify()`
- In pratica, la `notifyAll()` **sveglia tutti i thread bloccati** da una `wait()`, i quali poi, uno alla volta, acquisiranno il monitor di `obj` e proseguiranno la loro elaborazione

synchronized e **wait()** e **notify()**

- Si noti la differenza:
 - **synchronized** serve a garantire la **mutua esclusione**
 - solo un thread alla volta esegue una data parte di codice su un certo dato
 - **wait()** e **notify()** implementano un meccanismo di **notifica**
 - **wait()** serve a sospendersi in attesa di un evento
 - **notify()** serve a segnalare che l'evento si è verificato
- Non si tratta di tecniche interscambiabili, hanno scopi diversi!

Il comando **synchronized** (1)

- Un thread indica l'intenzione di acquisire un monitor con il comando **synchronized**

Comando ::= CmdSync | ...

CmdSync ::= synchronized (Espressione) Blocco

- Quando incontra questo comando, il thread prova ad acquisire il monitor associato all'oggetto ottenuto valutando l'*Espressione*
 - può anche sospendersi in attesa di acquisirlo
- Una volta ottenuto il monitor, esegue il *Blocco*
- All'uscita dal *Blocco*, rilascia il monitor

Il comando **synchronized** (2)

- Tipicamente, l'*Espressione* denota l'oggetto contenente i dati condivisi
 - il *Blocco* contiene istruzioni per accedere a questi dati
- Il comando **synchronized** garantisce che un solo thread alla volta possa eseguire il *Blocco* su quel monitor: si parla di mutua esclusione

```
public int get() {  
    synchronized (this) {  
        while (!disponibile) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
        disponibile = false;  
        //notifica il Produttore che il valore e' stato prelevato  
        notifyAll();  
        return contenuto;  
    }  
}
```

Vediamo alcune variazioni del produttore e consumatore

- Prima versione: `produttoreConsumatore.semplice` Classi:
 - Produttore, Consumatore, Contenitore e
 - Main
- Seconda versione: `produttoreConsumatore.classiAnonime` Classi:
 - Contenitore e
 - Main che genera Produttore e Consumatore senza definire le classi
- Terza versione: `produttoreConsumatore.buffer` Classi:
 - Buffer e
 - Main che genera Produttore e Consumatore senza definire le classi

Corso: Paradigmi di Programmazione

Paola Giannini

Input e Output in Java

Gli stream

- Uno **stream** è una sequenza ordinata di dati trasmessi
 - da una **sorgente**
 - verso una **destinazione**
- Uno stream è detto
 - di **lettura** se l'applicazione fa da destinazione
 - di **scrittura** se l'applicazione è la sorgente dei dati

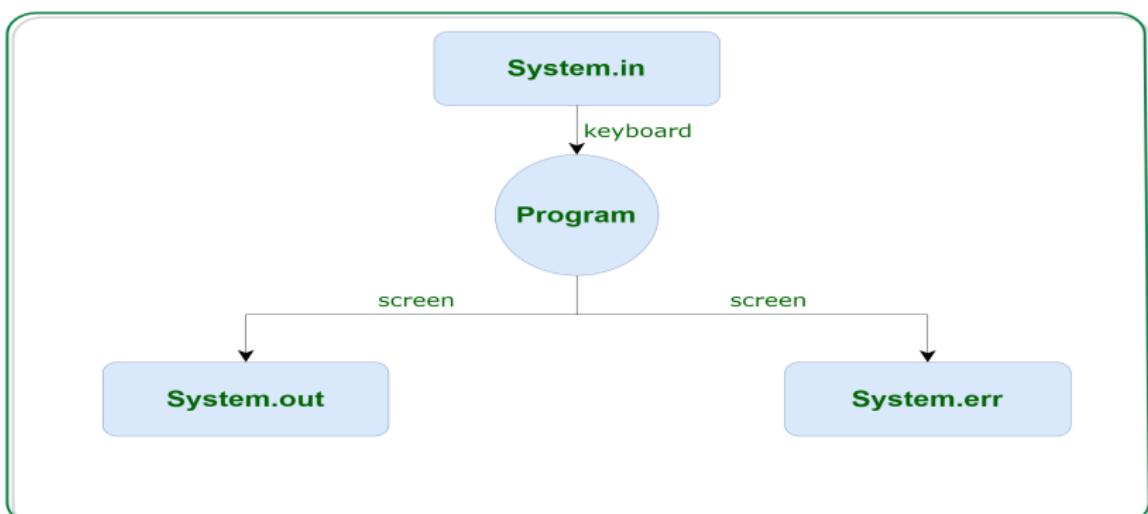
L'uso di uno stream prevede tre fasi principali

- **apertura** dello stream
- **lettura** o **scrittura** dei dati
- **chiusura** dello stream



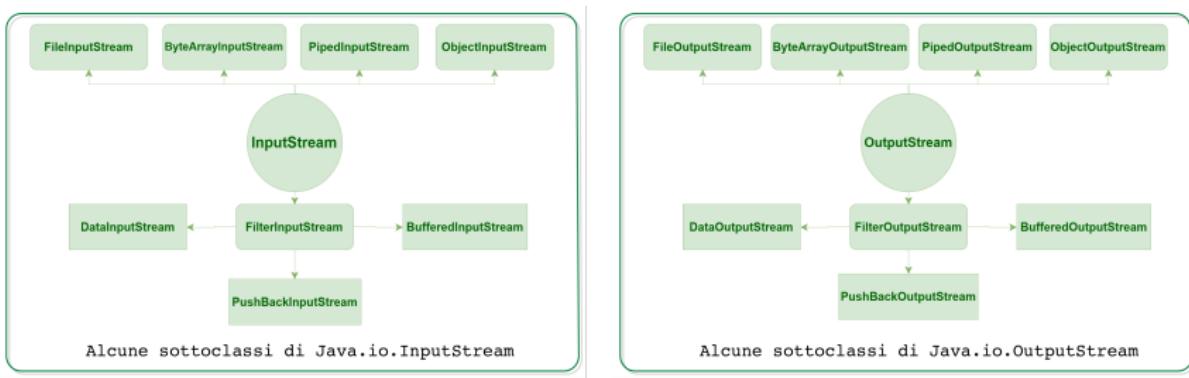
Gli stream standard della classe System

- La classe `java.lang.System` contiene alcuni campi statici che definiscono
 - lo standard input (`System.in` di tipo `InputStream`)
 - lo standard output (`System.out` di tipo `PrintStream`)
 - lo standard error (`System.err` di tipo `PrintStream`)
- Garantisce le funzionalità di base per l'interazione con l'utente attraverso il terminale
- Inoltre mette a disposizione i metodi accessori per ridirigere gli standard verso altri flussi



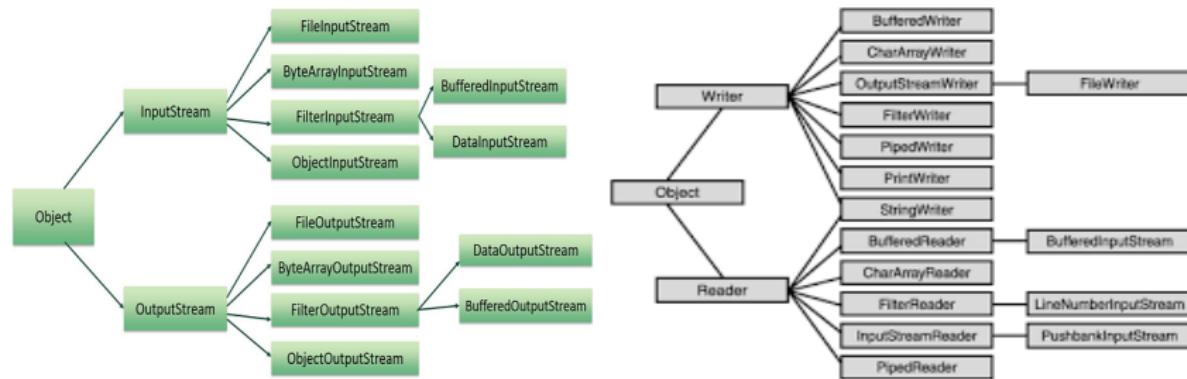
Tipi di Stream: operazioni

In dipendenza dalle operazioni ci sono due classi di Stream: quelli di input e quelli di output



Tipi di Stream: gestione di caratteri o di bytes

- In dipendenza dell'unità di dato che viene letto/scritto si distinguono: streams di bytes (8 bits) o di caratteri (permettono di leggere/scrivere i dati carattere per carattere).
- Inoltre alcune classi richiedono in input una classe che implementa l'interfaccia corrispondente e aggiungono nuovi metodi di lettura e/o scrittura.



Alcune eccezioni di input/output

- Problemi di accesso ai dati (sorgente/destinazione)
 - impossibile evitarli con controlli preventivi
 - le anomalie non dipendono dal programma
 - i malfunzionamenti possono verificarsi in ogni momento
- Eccezione **non controllata** `IOError` (eredita da `Error`)
 - modella problemi di I/O particolarmente gravi
- Tutte le altre eccezioni **sono controllate**
 - ereditano da `IOException`
 - `EOFException` (fine prematura di uno stream di lettura)
 - `FileNotFoundException` (impossibilità di aprire un certo file)
 - `ObjectStreamException` (es: errori su serializzazione di oggetti)

La classe java.io.File

- La classe File è una classe di utilità per la rappresentazione astratta di file system
 - astrazione indipendente dal sistema operativo
 - per ispezionare o manipolare il contenuto delle cartelle
 - per spostarsi tra le cartelle
 - per sapere la data dell'ultima modifica di un certo file
 - per sapere se un file può essere aperto in lettura o in scrittura

Costruttori e alcuni accorgimenti utili

```
File(File parent, String child)
File(String pathname)
File(String parent, String child)
File(URI uri)
```

- La classe File contiene

```
static String separator
```

che rappresenta la stringa di separazione per sottodirectory in modo indipendente dal sistema operativo (come sapete questi sono diversi in Windows o Unix/Mac)

Esempio: stampa ricorsiva dei nomi dei files

```
public static void stampaNomiFile(File corrente) {  
    if (corrente == null || !corrente.exists()) return;  
    else  
        if (corrente.isDirectory()) {  
            String[] elenco = corrente.list();  
            String padre = corrente.getAbsolutePath();  
            for (String nome : elenco) {  
                nome = padre + File.separator + nome;  
                stampaNomiFile(new File(nome));  
            }  
        }  
    else  
        if (corrente.isFile())  
            System.out.println(corrente.getName());  
}
```

La classe PrintStream

- PrintStream offre funzionalità avanzate per la scrittura sullo standard output e su file di testo
 - i metodi `print()` e `println()`
 - particolarmente versatili
 - in grado di gestire qualsiasi dato
 - non lanciano eccezioni controllate
 - il metodo `printf(String format, Object... args)`, in cui si può specificare il formato dei dati scritti (come per il C)
 - variadico
 - gli argomenti `args` vengono stampati nel formato specificato dalla stringa `format`
 - per stampe ben formattate (es: dati incolonnati, arrotondati)

I/O su file di testo

- Input e output di singoli caratteri su file di testo
 - classi FileReader e FileWriter
- Input e output bufferizzati e formattati
 - classi BufferedReader e BufferedWriter
 - encapsulano flussi di caratteri
 - metodi per leggere una riga alla volta (`readLine()`) o scrivere intere stringhe
 - classe PrintWriter
 - metodi `print()` e `println()` per trattare i tipi di dato primitivi e `printf()` per l'output formattato
 - i metodi di PrintWriter non lanciano eccezioni controllate: il metodo `checkError()` permette di accettare eventuali problemi

Esempio: lettura e scrittura su file

```
public static void leggiScriviFile(String inF, String outF) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(inF));
    PrintWriter out = new PrintWriter(new File(outF));
    String linea = in.readLine();
    int num = 1;
    while (linea != null) {
        String[] dati = linea.split(":");
        if (dati.length >= 3) {
            String cognome = dati[0].trim();
            String nome = dati[1].trim();
            String account = dati[2].trim();
            out.printf(" %s %s <%s@cli.di.unipi.it>,\n", nome, cognome, account);
        }
        else { System.out.println("Scarto linea n. " + num);}
        linea = in.readLine();
        num++;
    }
    in.close();
    out.printf("%n");
    out.close();
    System.out.println("Finito!");
}
```

Serializzazione

- La persistenza dei dati può essere ottenuta salvando le informazioni in formato testuale
 - modificabili con editor testuali
 - richiedono (de)codifica ad hoc dei dati (rischio di ridondanza, ambiguità, inconsistenza)
- La **serializzazione** (*object serialization*) offre un formato standard per la persistenza dei dati
 - basata su file di byte invece che di testo (**non editabili**)
 - garantisce che ogni oggetto è salvato al più una volta
 - risolve i problemi dovuti a ciclicità o condivisione di riferimenti

Salvare gli oggetti su file

- La classe `java.io.FileOutputStream` associa un flusso di byte a un file
- La classe `java.io.ObjectOutputStream` si occupa di **serializzare**, cioè tradurre gli oggetti in flussi di byte
 - ha un costruttore che permette di usare come flusso quello associato a un'istanza di `FileOutputStream`
 - metodi di scrittura per ogni tipo primitivo (es. `writeInt()`)
 - il metodo `writeObject(Object)` per serializzare oggetti
 - lancia l'eccezione non controllata `NotSerializableException` se l'argomento non implementa l'interfaccia `Serializable`

Leggere gli oggetti da file

- La classe `java.io.FileInputStream` associa un flusso di byte a un file
- La classe `java.io.ObjectInputStream` si occupa di **deserializzare**, cioè tradurre da flussi di byte a oggetti
 - ha un costruttore che permette di usare come flusso quello associato a un'istanza di `FileInputStream`
 - metodi di lettura per ogni tipo primitivo (es. `readInt()`)
 - il metodo `readObject()` per deserializzare oggetti
 - può lanciare alcune eccezioni controllate: ad esempio
`java.lang.ClassNotFoundException`,
`java.io.InvalidClassException`

Esempio: leggere e scrivere oggetti su file

```
public static void leggiScriviOggetti() throws FileNotFoundException, IOException,
    ClassNotFoundException {
    Persona p1 = new Persona("Paolo", "Rossi");
    // SCRIVE l'oggetto in un FILE BINARIO tramite la serializzazione
    ObjectOutputStream fbinarioOut = new ObjectOutputStream(new FileOutputStream("persone.dat"));
    fbinarioOut.writeObject(p1);
    fbinarioOut.flush();
    fbinarioOut.close();
    // LEGGE l'oggetto alunno salvato nel file, tramite la deserializzazione
    ObjectInputStream fin = new ObjectInputStream(new FileInputStream("persone.dat"));
    Persona p2 = (Persona) fin.readObject();
    // Visualizza l'oggetto sul monitor, sfruttando (implicitamente) il suo metodo toString()
    System.out.println(p2);
}

class Persona implements Serializable {
    private String nome;
    private String cognome;
    Persona(String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
    }
    public String toString() { return nome + " " + cognome; }
}
```

Sviluppare classi serializzabili

- Le istanze di una classe sono **serializzabili** solo se la classe implementa l'interfaccia `java.io.Serializable`: interfaccia vuota (senza metodi). Il controllo viene fatto a runtime, non durante la compilazione.
- Per default, `writeObject()` (ma si può ridefinire il comportamento) scrive sul flusso:
 - le informazioni per costruire le istanze della classe (la classe di appartenenza dell'oggetto e la segnatura della classe)
 - i valori di tutte le variabili distanza (della classe e delle eventuali **superclassi, che pure devono essere serializzabili**). Per default, i valori delle variabili di classe e delle variabili **transient** non vengono scritti sul flusso
- Per default, `readObject()` (ma si può ridefinire il comportamento) legge da un flusso:
 - le informazioni per costruire le istanze della classe (la classe di appartenenza dell'oggetto e la segnatura della classe)
 - fa l'**allocazione di memoria e inizializzazione**, poi esegue il **costruttore** e poi i campi della classe sono ripristinati al valore che è stato serializzato.

Un uso della serializzazione è fare una copia **deep** di un oggetto. Questo è complicato da fare a mano perchè la struttura degli oggetti può essere ciclica!



Corso: Paradigmi di Programmazione

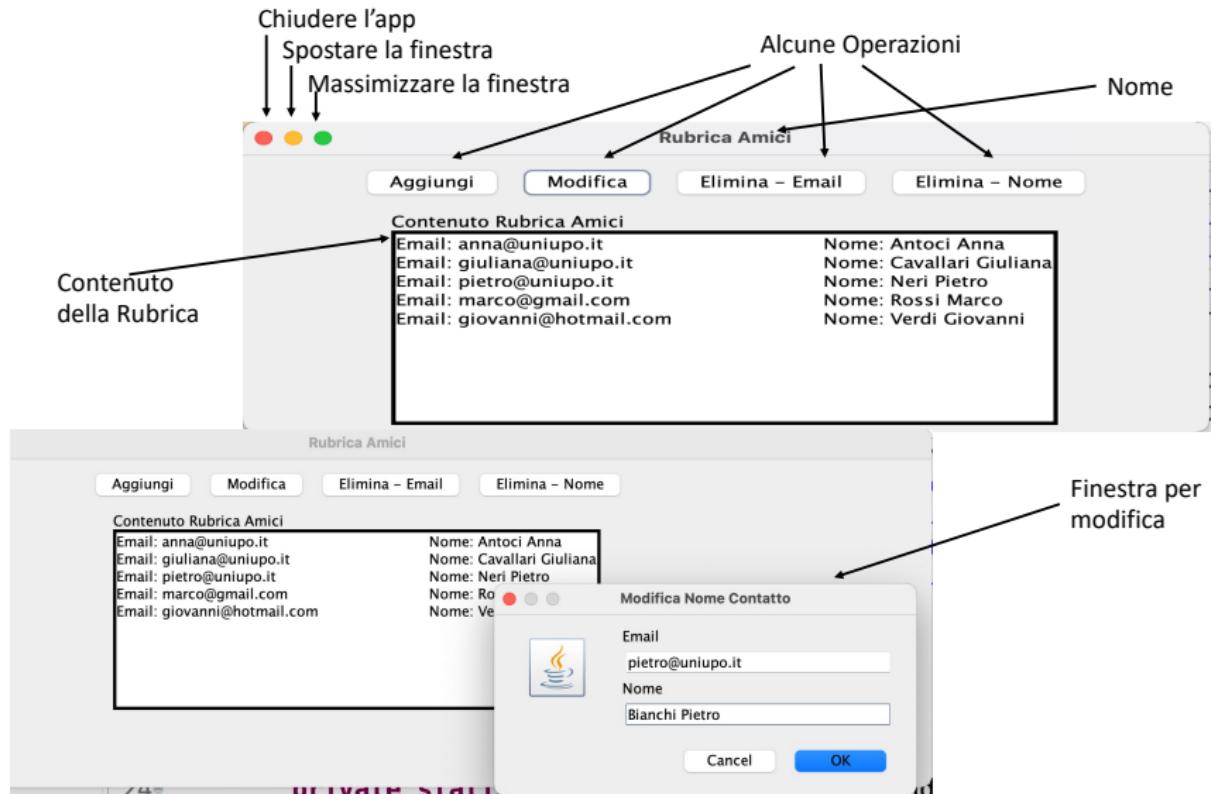
Paola Giannini

Model-View-Controller (MVC), Event Driven Programming (EDP)
(una Graphical User Interface (GUI) per Rubrica)

Cosa è una GUI

- Una GUI è una interfaccia ad un programma che fa uso di strumenti grafici come **bottoni, menù, disegni, finestre**,
- Facilita le operazioni di input e di visualizzazione del risultato dell'operazione
- Una GUI per la classe Rubrica potrebbe essere una finestra che permetta di fare
 - la creazione/eliminazione/modifica di un contatto
 - la ricerca/selezione di un contatto
 - la visualizzazione dei contatti presenti nella rubrica
- Se si gestiscono più rubriche ci può essere una ulteriore finestra che permette la creazione/cancellazione e selezione di una rubrica.

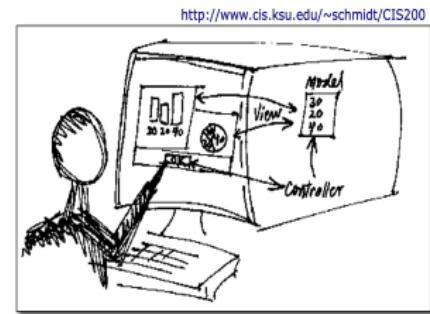
Gui (semplificata) per la Rubrica



Architettura Model View Controller (MVC)

Un programma si compone di

- un Modello (Model): dati e metodi che modellano il dominio dell'applicazione
- una Vista (View): rappresenta una "fotografia" dello stato interno del modello che ne facilita la lettura/interpretazione all'utente umano e l'esecuzione di operazioni sullo stesso
- un Controllore (Controller): controlla il flusso di informazioni fra la vista e modello

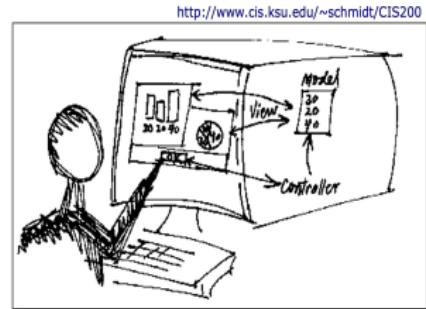


- Ha origine negli applicativi sviluppati in Smalltalk
- È stato utilizzato in Java per lo sviluppo delle componenti AWT/Swing

Funzionamento MVC

Come funziona il programma?

- L'utente agisce sulla vista agendo su una delle sue componenti di controllo (es. bottone)
- Il controllore è avvertito di tale evento ed esamina la vista per rilevarne le informazioni aggiuntive
- Il controllore invia tali informazioni al modello che effettua la computazione richiesta e aggiorna il proprio stato interno



- Il controllore richiede alla vista di visualizzare il risultato della computazione
- La vista interroga il modello sul suo nuovo stato interno e visualizza l'informazione all'utente

Vantaggi dell'architettura MVC

- In una applicazione tradizionale;
 - i frammenti di programma in cui si leggono gli input, quelli in cui si elaborano i risultati ed infine si restituiscono gli output sono inframezzati fra loro,
- Nell'architettura MVC:
 - L'applicativo è organizzato in parti più semplici e comprensibili (ogni parte ha le sue specifiche finalità)
 - Le classi che formano l'applicativo possono essere più facilmente riutilizzate
 - La modifica di una parte non coinvolge e non interferisce con le altre parti: maggiore flessibilità nella **manutenzione del software**.

Event-Driven Programming

- La programmazione event-driven è alla base della programmazione delle GUI
- Il nuovo tipo di input che deve essere trattato nella programmazione delle GUI è un'azione dell'utente quale la pressione di un bottone, il click del mouse, ecc.
- L'utente agendo sulla GUI genera degli **eventi** a cui il controllore deve reagire in maniera opportuna
- Si parla di **handling events**: processare/gestire gli eventi
- Il controllore che processa gli eventi è chiamato **event handler** o **event listener**
- Le informazioni sugli eventi, in Java, sono memorizzate in opportuni oggetti (**EventObject**)

Delegation Event Model

Il modello di programmazione in cui la gestione degli eventi è delegata agli ascoltatori coinvolge i seguenti oggetti:

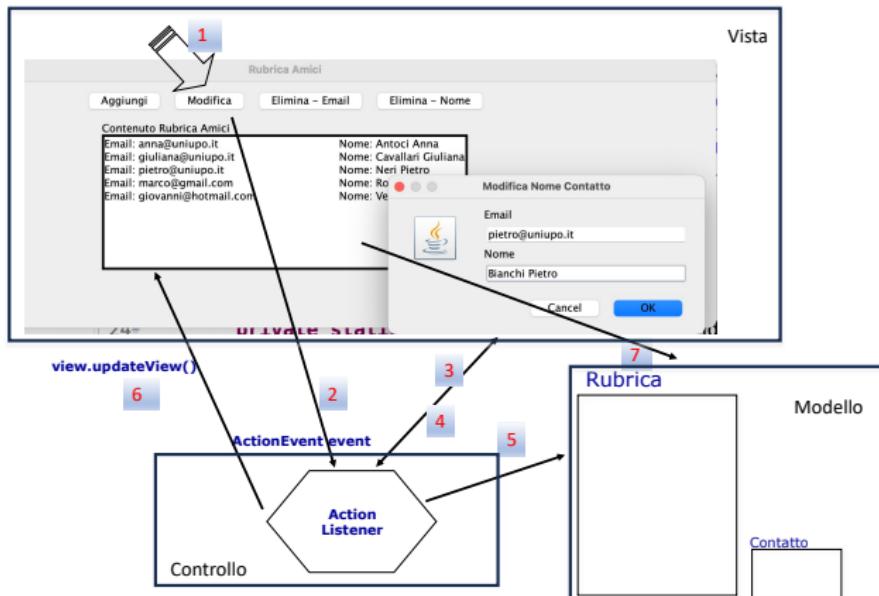
- Gli **eventi sono oggetti** contenenti le informazioni relative all'evento che li ha determinati
- L'**oggetto sorgente** nel caso della GUI è una componente grafica (es. un bottone),
- uno o più **oggetti ascoltatori** che si devono registrare con la componente grafica per ascoltare quel particolare evento.

Quando un evento è generato causa l'invocazione di un metodo degli oggetti ascoltatore che si erano registrati.



Eventi e chiamate di metodi per la Rubrica

- Il Sistema Operativo intercetta l'evento "click di un bottone" (1) e lo comunica all'interprete Java
- L'interprete determina la sorgente dell'evento, crea un ActionEvent e lo invia all'incaricato ActionListener (2)
- Il metodo actionPerformed del controllore è eseguito, e questo causa
 - la creazione della finestra di dialogo per l'operazione (3)
 - quando il tasto OK è premuto la lettura dei dati (4)
 - l'esecuzione dell'operazione sul modello (5)
 - la chiamata del metodo updateView della vista (6) e
- la vista si aggiorna interrogando il modello (7)



Risultato Finale

Rubrica Amici

Aggiungi Modifica Elimina – Email Elimina – Nome

Contenuto Rubrica Amici

Email: anna@uniupo.it	Nome: Antoci Anna
Email: giuliana@uniupo.it	Nome: Cavallari Giuliana
Email: pietro@uniupo.it	Nome: Bianchi Pietro
Email: marco@gmail.com	Nome: Rossi Marco
Email: giovanni@hotmail.com	Nome: Verdi Giovanni

Event-Driven Programming per GUI

Ogni **oggetto grafico** che può generare **eventi** (ad esempio un bottone `bot`) ha associata una lista di **oggetti in ascolto** (**listener objects**)

- un listener object `obj` è aggiunto alla lista di un oggetto `bot` tramite il metodo

```
bot.addActionListener(obj)
```

In generale, un componente grafico può

- generare più di un tipo di evento e
- vere **molti listener objects**

Un listener object può **ascoltare più componenti**.

- Quando un evento occorre, la lista viene scandita e
- a ogni listener object viene inviato il messaggio `actionPerformed`

java.awt e javax.swing (1)

Due packages principali per la gestione degli oggetti grafici: `java.awt` e `javax.swing` (non originale in Java, ma parte di una estensione X).

- `java.awt` (Abstract Windowing Toolkit) definisce le classi “base” (si appoggiano sul OS) per costruire e disegnare GUI.
- `javax.swing` costruita sulla precedente, fornisce oggetti grafici
 - “lightweight”, cioè non vengono mappati sulle componenti native del OS, ma si disegnano a partire da primitive grafiche semplificate
 - possono avere “look and fee” diversi
 - anche componenti quali bottoni ed etichette sono contenitori per cui possono contenere ad esempio gif

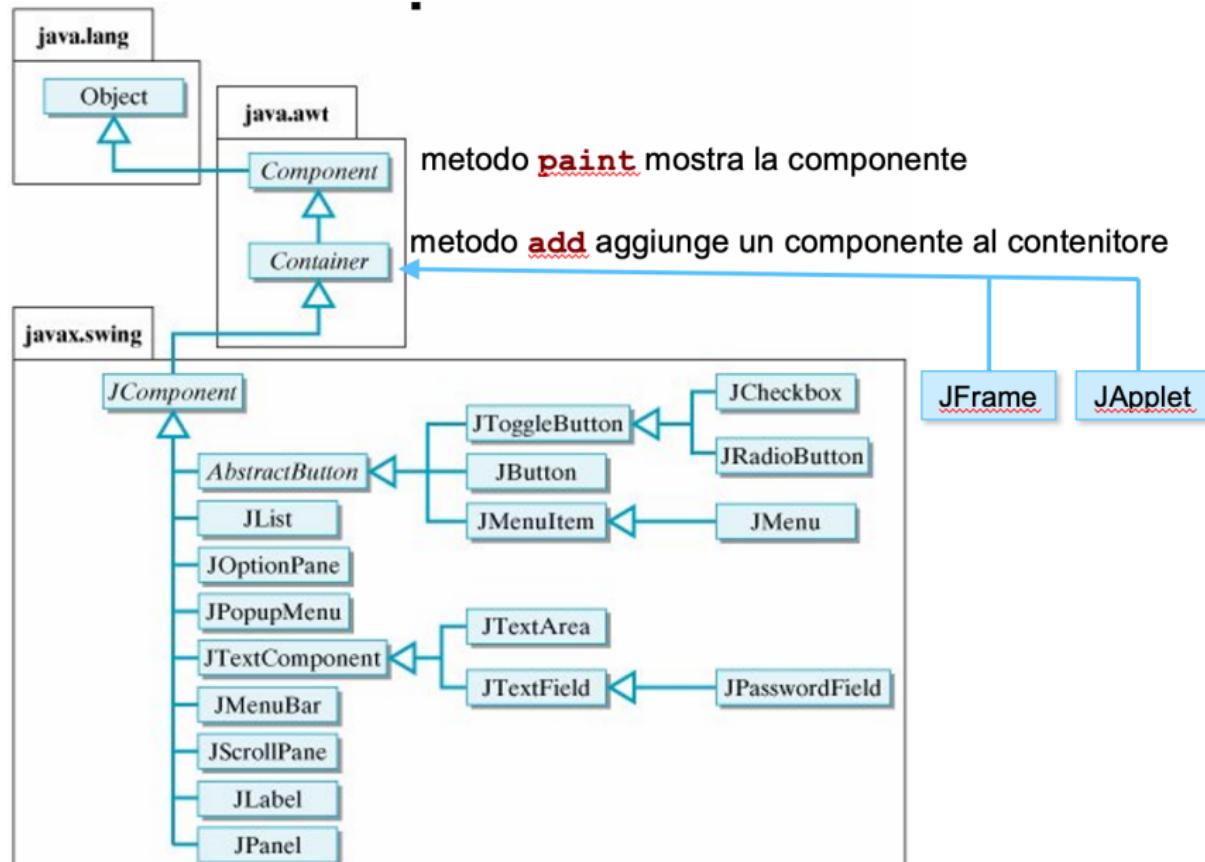
In generale, a fronte di una classe in `java.awt` esiste luna classe in `javax.swing`. Es. per `Panel`, `Button` in `awt` esistono `JPanel`, `JButton` in `javax.swing`

java.awt e javax.swing (2)

Una GUI fatta con AWT/Swing è costituita da

- **Componenti** (Component): oggetti che possono avere una posizione e una dimensione nello schermo e nei quali possono occorrere eventi. Ad esempio Label (JLabel), Button (JButton), ecc.. sono componenti di AWT (Swing).
- **Contenitori** (Container): componenti che possono contenere al loro interno altre componenti come, ad esempio, Panel (JPanel), Frame (JFrame)
- Per i Contenitori è specificato un layout: cioè come le componenti inserite nel contenitore dovranno essere posizionate.

Struttura delle componenti grafiche



Componenti top-level (contenitori di applicazione)

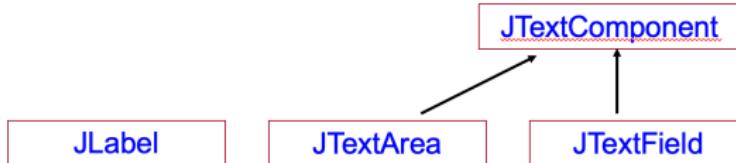
Al top level di una applicazione possiamo avere un `JFrame` oppure una `JApplet`

- `JFrame` contenitore che useremo per la nostra applicazione
 - può contenere `JMenuBar`, `JMenu`, ecc..
 - deve essere dimensionato e reso visibile
- `JApplet` eseguite nel browser (introdotte per far migrare codice eseguibile)
 - Eseguite in un contesto che non è quello in cui sono state create.
 - Hanno un protocollo di esecuzione (sequenza di chiamata di metodi) fissato
 - Il modello originale di sicurezza di Java prevedeva che le applets girassero in un “sandbox” (nessun accesso a risorse locali)
 - Il modello di sicurezza di Java nuovo permette al client di specificare le politiche di sicurezza (accesso in lettura/scrittura ecc..)

Eventi nella GUI

- Ogni componente grafica è in una area della GUI (ha un rettangolo che la contiene).
- Una componente **ha il focus** (quando il mouse passa sull'area in cui è contenuta). Solo una componente alla volta può avere il focus.
- La componente che ha il focus è il **recipiente degli eventi** del Mouse e/o della Keyboard.
- Se la componente ha registrato un listener per l'evento, il metodo appropriato del listener viene chiamato passando le informazioni L'oggetto che viene passato al metodo che gestisce l'evento è MouseEvent e contiene
 - Informazioni sull'oggetto che ha generato l'evento e inoltre
 - registra se i tasti shift oppure ctr erano premuti

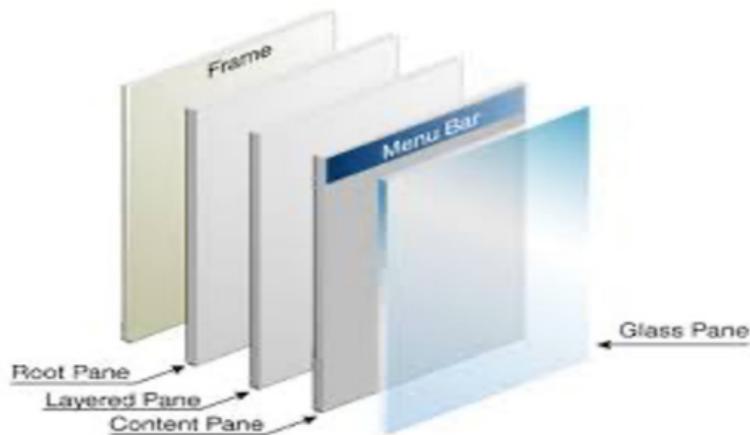
Componenti di testo



- Tutte le componenti grafiche generano gli eventi di `JComponent` (cambio di dimensione/posizione/visibilità, focus, movimento mouse, ...)
 - `JLabel` può aver associato un testo, ma non può essere usata come input (il valore del testo deve essere specificato nel programma), non genera eventi propri
 - `JTextField` è una riga di testo che può essere resa editabile. Quando si va a capo è generato un evento di tipo `ActionEvent` (come per il click su un Bottone)
 - `JTextArea` può contenere più righe (non viene generato un evento quando si va a capo)
- Come gli oggetti sono mostrati nella GUI è determinato dal metodo `paint` che ha una definizione standard in ogni componente. Ad esempio:
 - per `JLabel` e `JButton` disegna un rettangolo e ci mette dentro la stringa associata
 - per `Container` chiama il metodo `paint` di ogni sua componente

Il contenitore top-level

- L' applicazione (contenitore con bottoni, etichette, ecc..) sarà contenuta in un JFrame.
- La componente JFrame è costituita da diversi strati. Quello che a noi interessa (cioè dove aggiungeremo la nostra vista è ContentPane)
- Aggiungeremo al ContentPane del JFrame un oggetto (JPanel) che contiene la vista dell'applicazione.



Il contenitore JPanel e Layouts

- JPanel è un contenitore che permette di organizzare componenti.
- Il layout specifica come verranno aggiunti i contenuti (componenti o contenitori), metodi add.
- Il più semplice è FlowLayout aggiunge da sinistra a destra (default per JPanel).
- Molto usato è BorderLayout specifica che le componenti interne sono posizionate in 5 zone e add aggiunge a una zona specifica.

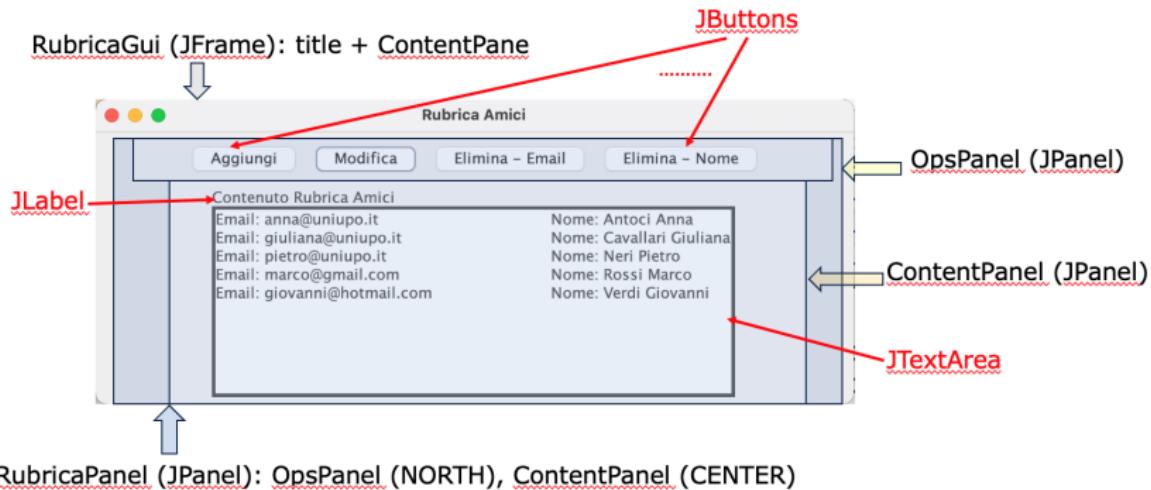


es. per aggiungere in CENTER la componente c: `add(c, BorderLayout.CENTER)`

- Il default per ContentPane è BorderLayout.

Gli oggetti della GUI (vista)

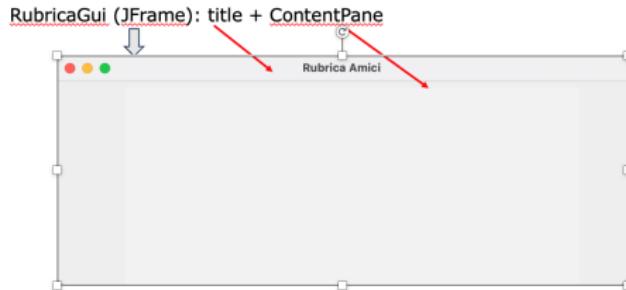
- I contenitori e le componenti dell'interfaccia (in rosso Component, in nero Container)



- Costruiremo la GUI top-down: dagli elementi più esterni

La classe RubricaGUI

- Iniziamo dal JFrame



```
public class RubricaGui extends JFrame{  
    public RubricaGui(Rubrica model) {  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setBounds(100, 100, 450, 300);  
        setTitle("Rubrica " + model.getNome());  
  
        JPanel rubricaPanel= new RubricaPanel(model);  
        setContentPane(rubricaPanel);  
        setVisible(true);  
    }  
}
```

- JFrame potrebbe avere una barra di menu.

La classe RubricaPanel

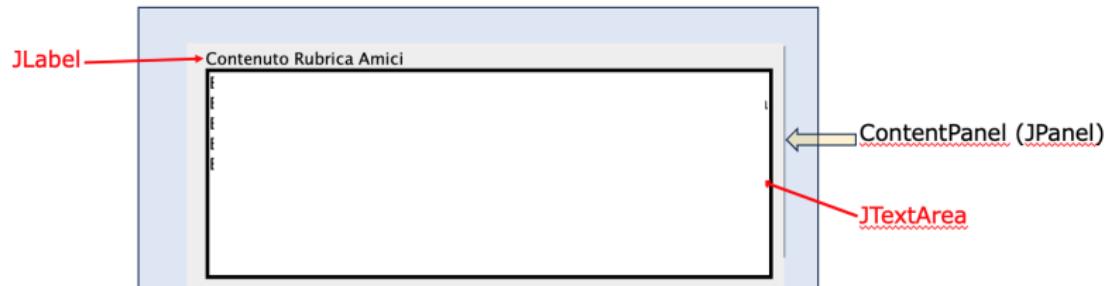
- RubricaPanel contiene il pannello delle operazioni (contiene i bottoni) e quello in cui si mostra il contenuto.
- Dobbiamo anche creare il Controllore (il listener dei button per passarlo al pannello delle operazioni).



```
public class RubricaPanel extends JPanel{
    public RubricaPanel(Rubrica model) {
        // setta il layout a BorderLayout
        ContentPanel contenutoRubrica=new ContentPanel(model);
        ControlloRubrica controllo = new ControlloRubrica(contenutoRubrica,model);
        OpsPanel operazioniRubrica = new OpsPanel(controllo);
        // Aggiungere le componenti al pannello nelle posizioni giuste
    }
}
```

La classe ContentPanel (1)

- In questo pannello mettiamo una etichetta e una JTextArea nella quale mostriamo il contenuto della rubrica.



```
public class ContentPanel extends JPanel {  
    private Rubrica model;  
    private JTextArea rubrica;  
    public ContentPanel(Rubrica model) {  
        this.model = model;  
        setLayout(new BorderLayout());  
        rubrica = new JTextArea(10,20);  
        rubrica.setBorder(BorderFactory.createLineBorder(Color.BLACK,3));  
        JLabel label = new JLabel("Contenuto Rubrica " + model.getNome());  
        add(label, BorderLayout.NORTH);  
        add(rubrica, BorderLayout.CENTER);  
        ....  
    }  
    ....  
}
```

La classe ContentPanel (mostrare la rubrica)

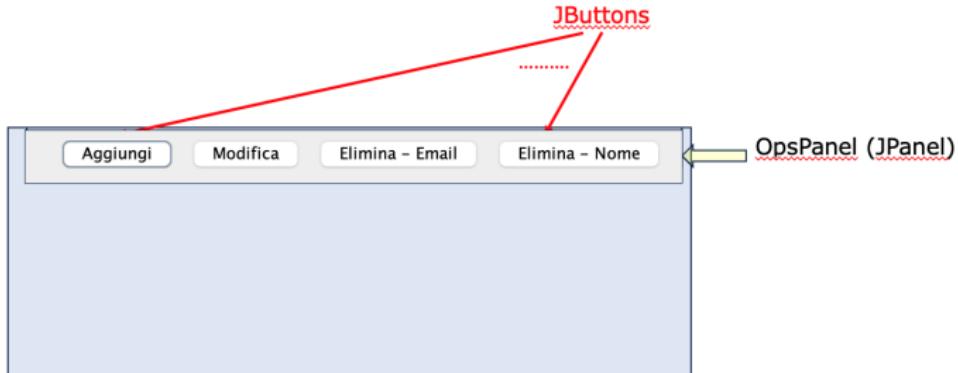
- Per mostrare la rubrica si usa il metodo `updateView` che prende i dati dalla rubrica (con `toString()`) e li mostra nella `TextArea`.



```
public class ContentPanel extends JPanel {  
    private Rubrica model;  
    private JTextArea rubrica;  
    public ContentPanel(Rubrica model) {  
        this.model = model;  
        setLayout(new BorderLayout());  
        rubrica = new JTextArea(10,20);  
        rubrica.setBorder(BorderFactory.createLineBorder(Color.BLACK,3));  
        JLabel label = new JLabel("Contenuto Rubrica " + model.getNome());  
        add(label, BorderLayout.NORTH);  
        add(rubrica, BorderLayout.CENTER);  
        updateView();  
    }  
    public void updateView() {  
        rubrica.setText(model.toString());  
    }  
}
```

La classe OpsPanel

- In questo pannello mettiamo i bottoni che devono far eseguire le operazioni sul modello (la rubrica).
- Ai bottoni dobbiamo aggiungere i listener (qua cattureremo solo l'evento Action!)



```
public class OpsPanel extends JPanel {  
    public OpsPanel(ControlloRubrica controllo) {  
        setLayout(new FlowLayout());  
        JButton addContatto = new JButton("Aggiungi"); //aggiungiamo i bottoni  
        //.....  
        addContatto.addActionListener(controllo); //aggiungiamo il listener di evento Action  
        //.....  
        add(addContatto); // aggiungiamo i bottoni al pannello (da sinistra a destra)  
        //.....  
    }  
}
```

- Per il mouse c'è anche MouseEvent che permette un controllo più raffinato di Action. Oltre al click permette di catturare la pressione e/o rilascio del mouse e altro..



Il controllo

- Gli eventi che catturiamo sono di tipo Action per cui dobbiamo implementare l'interfaccia ActionListener che prevede di implementare il metodo

```
public void actionPerformed(ActionEvent e)
```

- Per prendere input dall'utente definiamo la classe DialogoContatto il cui metodo getInputs ritorna un array di String il primo elemento è l'email e il secondo il nome.

```
public class ControlloRubrica implements ActionListener {  
    private Rubrica model;  
    private ContentPanel view;  
    public ControlloRubrica(ContentPanel view, Rubrica model) {  
        this.model = model; this.view = view;  
    }  
    public void actionPerformed(ActionEvent e) {  
        JButton source = (JButton) e.getSource(); // il cast !!  
        if (source.getText().equals("Aggiungi")) { // da quale bottone arriva l'azione?  
            String[] inputs = new DialogoContatto().getInputs("Aggiungi Contatto");  
            if (inputs != null) {  
                try { model.aggiungi(inputs[0], inputs[1]); }  
                catch (RubricaException e) {System.out.println(e.getMessage());}  
            }  
        }  
        else if (source.getText().equals("Modifica")) {  
            .....  
        }  
        else {  
            .....  
        }  
        view.updateView(); //dopo l'operazione la vista mostra la rubrica modificata.  
    }  
}
```

JOptionPane

- Classe swing che viene utilizzata per fornire finestre di dialogo standard. Noi la useremo per avere un dialogo di input.
- Più semplice di doversi creare una propria sottoclasse di JDialog, ma anche più limitata!
- La chiamata del metodo

```
JOptionPane.showConfirmDialog(.....)
```

crea una finestra che ha

- un titolo e
- nella quale si possono inserire componenti (es: JTextField per inserire e JLabel per mostrare testo)
- che ha bottoni per comunicare che c'è (o meno) informazione da recuperare.

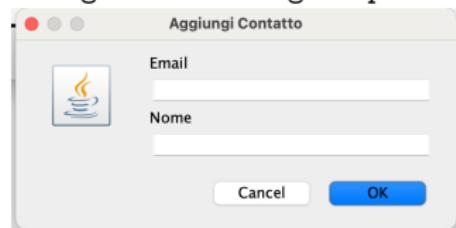
Inoltre ha un bottone OK e Cancel che provocano il ritorno dal metodo
`JOptionPane.showConfirmDialog(.....)` con l'opzione scelta.

La classe DialogoContatto

- Per prendere input dall'utente definiamo la classe DialogoContatto il cui metodo getInputs ritorna un array di String il primo elemento è l'email e il secondo il nome.

```
public class DialogoContatto {  
    private JTextField nome, email;  
    private JComponent[] inputs;  
    public DialogoContatto() {  
        // sicerano le componenti grafiche che andranno nella finestra di dialogo  
        nome=new JTextField(20);email=new JTextField(20);  
        inputs=new JComponent[]{new JLabel("Email"),email,new JLabel("Nome"),nome,};  
    }  
    public String[] getInputs(String msg) {  
        String[] res=new String[2];  
        //si crea la finestra di dialogo  
        int result= JOptionPane.showConfirmDialog(null,inputs,msg,OptionPane.CANCEL_OPTION);  
        //bloccato fino a quando c'e' un evento Action (ad esempio si preme OK o Cancel)  
        if (result == JOptionPane.OK_OPTION) { // se si e' premuto OK  
            // mettiamo le stringhe nei 2 JTextField nell'array che viene ritornata  
            res[0]=email.getText(); res[1]=nome.getText(); return res;  
        } else { return null; }  
    }  
}
```

La finestra di dialogo:`new DialogoContatto().getInputs("Aggiungi Contatto")`



JMenu e JMenuItem

- Altro modo per eseguire operazioni è usare dei menu.
- La componente JFrame (ma anche JWindow che è simile ma non ha chiusura/spostamento/massimizzazione) può avere unMenuBar
- AlMenuBar si possono aggiungere JMenu che possono contenere JMenuItem
- La selezione di un JMenuItem provoca un evento Action (come il click del mouse).
- Quindi per aggiungere un'azione si aggiunge al JMenuItem un oggetto che implementa ActionListener cioè definisce il metodo actionPerformed.
- Vediamo un esempio che mostra anche un utile componente per fare il browsing del file system.