

Programmazione Orientata agli Oggetti

Classi astratte

Interfacce

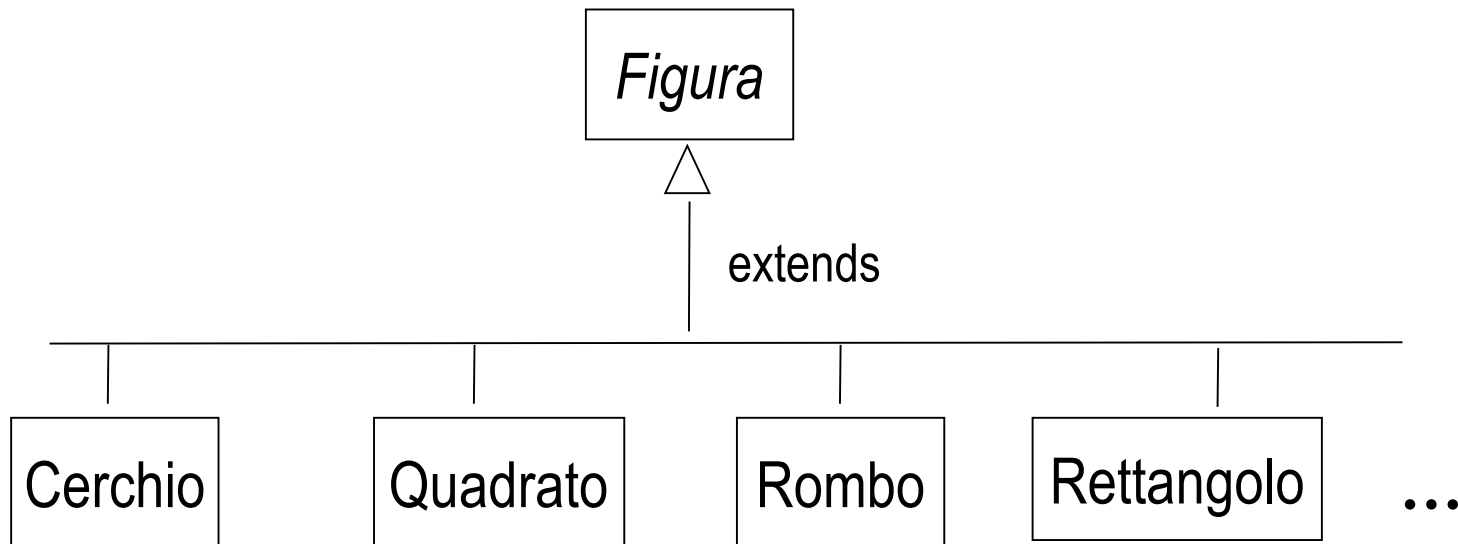
Classi stringhe

Gestione eccezioni

Libero Nigro

Una gerarchia di classi per figure geometriche piane

- Si considerano le comuni figure piane come cerchio, quadrato, rombo, trapezio etc.
- Volendo organizzare le figure in modo da facilitarne l'utilizzo nelle applicazioni, si può riflettere che tutte posseggono almeno una dimensione, es. il raggio per il cerchio, il lato per il quadrato o il rombo, la base e l'altezza per un rettangolo e così via
- Per “imparentare” le figure si può concepire una classe base **Figura** che poi ogni figura particolare può estendere e specializzare. In **Figura** si può introdurre una dimensione (double) e i metodi che certamente hanno senso su tutte le figure



- Identificare una gerarchia di classi come quella di cui si sta discutendo è sempre un **fatto importante**: infatti si può introdurre nella classe base (qui Figura) tutti quegli elementi (attributi e metodi) **comuni** a qualunque erede. In questo modo si evitano ridondanze e si garantisce ad ogni classe derivata di possedere i “connotati” di appartenenza ad una stessa “famiglia”
- Si rifletta ora che prevedendo una dimensione (cioè un lato) nella classe Figura, non si sa bene cosa essa voglia dire. Per un cerchio si tratterà del suo raggio, per un quadrato del suo lato, per un rettangolo magari la sua base etc. Quindi metodi come perimetro() ed area() previsti in Figura non si possono dettagliare in quanto manca l'informazione su come interpretare la figura
- Si dice che una classe come Figura è astratta (**abstract**) proprio perchè ancora incompleta. Spetta poi alle classi eredi concretizzare tutti quegli aspetti previsti in Figura ma al momento astratti. Segue una specifica della classe astratta Figura

```
package poo.figure;  
public abstract class Figura{  
    private double dimensione;  
    public Figura( double dim ){ this.dimensione=dim; }  
    protected getDimensione(){ return dimensione; }  
    public abstract double perimetro();  
    public abstract double area();  
} //Figura
```

- Una classe astratta come Figura non è istanziabile, ossia non si possono creare oggetti della classe. Allora a cosa serve ? Serve come base per progettare classi eredi
- Per etichettare che una classe è astratta si premette al nome class il modificatore **abstract**.
- In una classe astratta almeno un metodo deve essere astratto.
- Una classe erede di Figura è concreta se implementa (ne fornisce cioè il corpo) **tutti** i metodi abstract. Se qualche metodo rimane ancora astratto, anche la classe erede è astratta e spetta ad un ulteriore erede implementare i rimanenti metodi abstract etc.
- Si nota che in una classe astratta possono essere presenti campi dati (es. dimensione) e metodi concreti. Ad esempio getDimensione() e setDimensione() sono concreti.

```
package poo.figure;
import poo.util.Mat;
public class Cerchio extends Figura{//esempio di classe concreta
    public Cerchio( double raggio ){ super(raggio); }
    public Cerchio( Cerchio c ){ super(c.getDimensione()); }
    public double getRaggio(){ return getDimensione(); }
    public double perimetro(){
        return 2*Math.PI*getDimensione();
    }//perimetro
    public double area(){
        double r=getDimensione();
        return r*r*Math.PI;
    }//area
}
```

```

public String toString(){
    return "Cerchio: raggio="+getDimensione();
}
//toString
public boolean equals( Object x ){
    if( !(x instanceof Cerchio) ) return false;
    if( x==this ) return true;
    Cerchio c=(Cerchio)x;
    return
        Mat.sufficientementeProssimi(
            this.getDimensione(), c.getDimensione() );
}
//equals
}
//Cerchio

```

Cerchio è classe concreta in quanto implementa tutti i metodi astratti di Figura.

Essendo privato il campo dimensione di Figura, si è fatto ricorso al metodo `getDimensione()` per accedervi da dentro Cerchio.

Il metodo `equals()` necessariamente è peculiare di ogni classe erede, e per questa ragione non è stato previsto in Figura. Similmente per il metodo `toString()`. In altre situazioni può essere invece conveniente anticipare nella super classe un'implementazione dei metodi `equals()`, `hashCode()` e `toString()`. Segue un'altra classe concreta: Rettangolo

```
package poo.figure;
import poo.util.Mat;
public class Rettangolo extends Figura{
    protected double altezza;
    public Rettangolo( double base, double altezza ){//PRE: dati corretti
        super( base );
        this.altezza=altezza;
    }
    public Rettangolo( Rettangolo r ){
        super( r.getDimensione() );
        this.altezza=r.altezza;
    }
    public double getBase(){ return getDimensione(); }
    public double getAltezza(){ return altezza; }
    public double perimetro(){
        return 2*getDimensione()+2*altezza;
    }//perimetro
    public double area(){
        return getDimensione()*altezza;
    }//area
}
```

```

public String toString(){
    return "Rettangolo: base="+getDimensione()+" altezza="+altezza;
} //toString
public boolean equals( Object x ){
    if( !( x instanceof Rettangolo) ) return false;
    if( x==this ) return true;
    Rettangolo r=(Rettangolo)x;
    return Mat.sufficientementeProssimi( r.getDimensione(), this.getDimensione() )
        &&
        Mat.sufficientementeProssimi( r.getAltezza(), this.altezza );
} //equals
} //Rettangolo

```

Esercizio: programmare altre classi eredi di Figura come: Quadrato, Rombo, Triangolo, TrapezioIsoscele, etc. La classe Triangolo potrebbe esportare un metodo per conoscere il tipo di triangolo etc.

- Si mostra un metodo areaMassima() che riceve un array di figure e scrive in uscita l'area massima e il tipo di figura avente l'area massima. L'array contiene oggetti Figura (la super classe)

```
public static void areaMassima( Figura []f ){  
    double am=0; Figura fam;  
    for( int i=0; i<f.length; i++ ){  
        double a=f[i].area(); //dynamic binding  
        if( a>am ){ am=a; fam=f[i]; }  
    }  
    System.out.println("La figura "+fam+" ha area massima="+am);  
} //areaMassima
```


Il metodo areaMassima potrebbe appartenere ad una classe con il main come segue:

```
public class Applicazione{  
    public static void areaMassima(){...}  
    public static void main( String []args ){  
        ...  
        Figura []a={ new Cerchio(4), new Rettangolo(2,5), ... };  
        areaMassima( a );  
        ...  
    }//main  
}//Applicazione
```

Ereditarietà singola e interfacce

- Anche se non è possibile per una classe ereditare simultaneamente da più di una super classe, Java introduce un meccanismo per “simulare” l’eredità multipla: le **interfacce**
- Una classe può *estendere* **una** sola classe ma può *implementare* **zero, una o più** interfacce
- Un’interfaccia (interface) è una raccolta di intestazioni (signature) di metodi. Si tratta di definizioni astratte di metodi ma senza la parola `abstract`
- Un’interfaccia, così come una classe astratta, non è istanziabile
- Una classe che implementi un’interfaccia **dovrebbe** fornire **un’implementazione** di **tutti** i metodi definiti nell’interfaccia, diversamente la classe va battezzata come astratta

L'interfaccia Comparable

```
public interface Comparable{  
    int compareTo( Object x );  
} // Comparable
```

Per massima generalità, compareTo() lavora su Object

compareTo(x) ritorna <0, ==0, >0 a seconda che l'oggetto this sia rispettivamente minore, uguale o maggiore di x

Nota: il modificatore public davanti ai metodi di una interface è opzionale. Esso è sottinteso se assente.

Razionali comparabili

```
package poo.razionali;
import poo.util.Mat;
public class Razionale implements Comparable{
    ...//come prima
    public int compareTo( Object x ){
        Razionale r=(Razionale)x;
        int mcm=Mat.mcm(this.denominatore,r.denominatore);
        int n1=(mcm/this.denominatore)*this.numeratore;
        int n2=(mcm/r.denominatore)*r.numeratore;
        if( n1<n2 ) return -1;
        if( n1>n2 ) return 1;
        return 0;
    }//compareTo
}//Razionale
```

Effetti di *implements*

- Dal fatto che Razionale estende (implicitamente) Object, discende che i razionali sono *anche* di tipo Object.
- Dal fatto che Razionale implementa Comparable, deriva che gli oggetti razionali sono *anche* comparabili, ossia di tipo Comparable (aumento del polimorfismo).
- Si dice che Comparable è *super tipo* di Razionale che ne è un *sub tipo*, similmente alla relazione di ereditarietà.
- Un array di Comparable è dunque un array di oggetti sui quali è definito il criterio di confronto.
- Comparable è un'interfaccia già presente in Java (è definita in java.lang).

Una classe di servizio

```
package poo.util;
public final class Array{//versione completa fornita a parte
    private Array(){
    public static void selectionSort( Comparable []v ){
        for( int j=v.length-1; j>0; j-- ){
            int indMax=0;
            for( int i=1; i<=j; i++ )
                if( v[i].compareTo(v[indMax])>0 ) indMax=i;
            //scambia v[indMax] con v[j]
            Comparable park=v[j]; v[j]=v[indMax];
            v[indMax]=park;
        }
    }//selectionSort
    public static void bubbleSort( Comparable []v ){...}
    public static int ricercaBinaria( Comparable []v, Comparable x){...}
    ...
}//Array
```

Un'implementazione di bubbleSort in poo.util.Array

```
public static void bubbleSort( Comparable []v ){
    int ius=0;//inizializzazione fittizia
    for( int j=v.length-1; j>0; j=ius ){
        int scambi=0;
        for( int i=0; i<j; i++ )
            if( v[i].compareTo(v[i+1])>0 ){
                //scambia v[i] con v[i+1]
                Comparable park=v[i];
                v[i]=v[i+1]; v[i+1]=park;
                scambi++; ius=i;//indice ultimo scambio
            }
        if( scambi==0 ) break;
    }
    }//for esterno
};//selectionSort
```

Ordinamento di razionali comparabili

In un main:

...

```
Razionale []v={ //esempio  
    new Razionale(2,3), new Razionale(4,7),  
    new Razionale(2,8), new Razionale(3,9) };
```

```
Array.bubbleSort( v );  
for( int i=0; i<v.length; i++ )  
    System.out.println( v[i] );
```

Per scrivere l'array su output si può anche ricorrere al metodo di servizio
`java.util.Arrays.toString(array)`:

```
System.out.println( java.util.Arrays.toString( v ) );
```

che può essere invocato anche prima di ordinare l'array per visualizzarlo nello stato iniziale disordinato

- L'uso dell'interfaccia Comparable rende possibile approntare una classe di utilità come Array che esporta i più comuni algoritmi di ordinamento e ricerca (lineare e binaria). Diverse varianti sono disponibili di uno stesso metodo (overloading): ad es. oltre a selectionSort che accetta un array di Comparable, c'è una versione che accetta un array di int e un'altra che accetta un array di double. Inoltre, altre tre versioni sono presenti che accettano l'array e la sua dimensione specifica (size) così consentendo di lavorare su array incompletamente riempito
- Questo modo di operare, come si vedrà nel seguito, è ampiamente sfruttato dalla libreria di Java (API)
- Per avvalersi di un metodo qualsiasi di ordinamento di Array, è sufficiente che una classe applicativa implementi Comparable
- In Java, quando una classe implementa Comparable, si dice che i suoi oggetti dispongono *dell'ordinamento naturale*
- Le interfacce possono essere costruite anche per estensione (**extends**). Se l'interfaccia I2 estende I1, allora banalmente in I2 si ritrovano tutte le intestazioni di metodi di I1 più quelle previste da I2

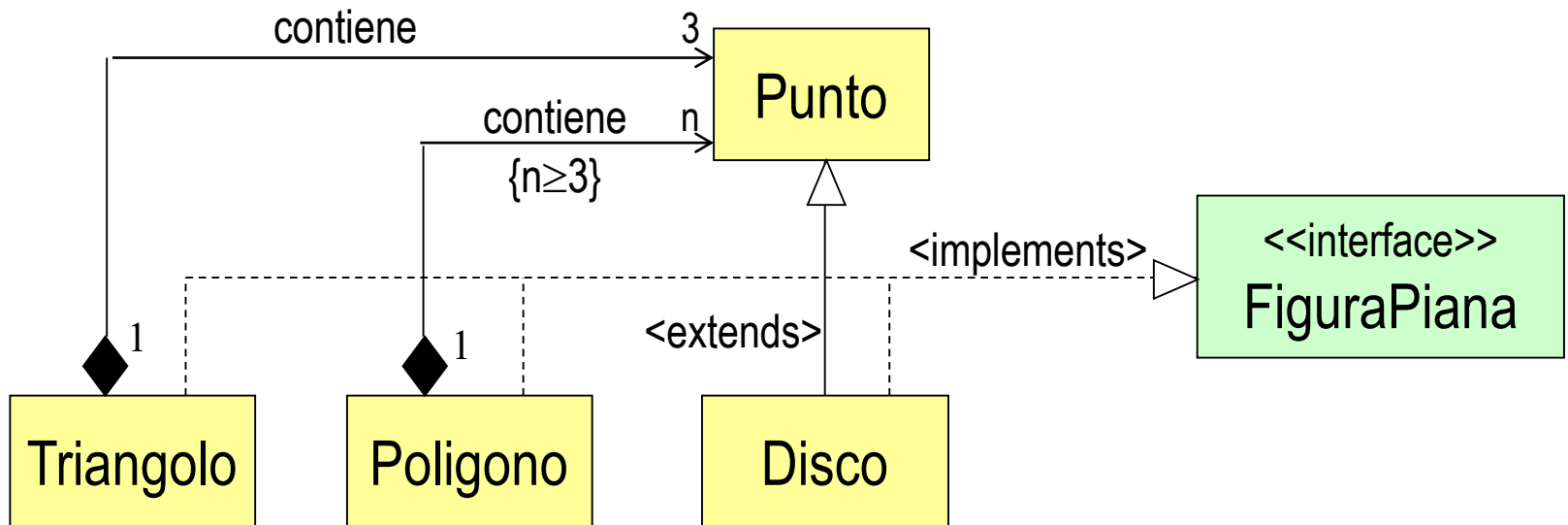
Regole di “buon” progetto di una classe Java

- Alla luce delle conoscenze sin qui acquisite, si può dire che il progetto di una classe, per generalità, dovrebbe
 - prevedere il metodo *boolean equals(Object x)*
 - prevedere il metodo *String toString()*
 - prevedere il metodo *int hashCode()* che ritorna un int identificativo unico dell'oggetto. Se due oggetti sono uguali secondo *equals()* allora il loro *hashCode* **deve** essere uguale. Per definire l'*hashCode* occorre prestare attenzione a ciò che identifica un oggetto, es. per una persona si potrebbe basare l'*hashCode* sul campo codice fiscale, per uno studente si potrebbe considerare la matricola etc. Per esempi si rimanda al seguito del corso
 - implementare l'interfaccia *Comparable* e dunque il metodo *compareTo*, se si prevede che gli oggetti debbano essere assoggettati ad ordinamento o comunque a confronti (es. per ragioni di ricerca)

Un altro esempio d'uso di interfaccia

- Un'interfaccia consente di accomunare classi che diversamente resterebbero isolate e dunque trattate ad hoc
- Si consideri una semplice gerarchia: dalla classe Punto si deriva una classe Cerchio, che in più aggiunge il raggio. Ricordiamo che in precedenza abbiamo definito una classe Triangolo che contiene tre punti, una classe Poligono (supposto convesso) che contiene $n \geq 3$ punti. Triangolo e Poligono non estendono Punto ma contengono (has-a) punti. Ovviamente queste tre classi Cerchio, Triangolo e Poligono non condividono nulla (eccetto che derivano da Object)
- In questa situazione, potrebbe essere conveniente introdurre una interfaccia **FiguraPiana** con i due metodi per calcolare perimetro ed area. Imponendo a Cerchio, Triangolo e Poligono di implementare questa stessa interfaccia, di fatto si ottiene di “imparentarle” e di considerarle in modo omogeneo, ad esempio conservandone oggetti in un array di FiguraPiana

Un diagramma di classi UML



Codice Java

```
package poo.geometria;  
public interface FiguraPiana{  
    double perimetro();  
    double area();  
}//FiguraPiana
```

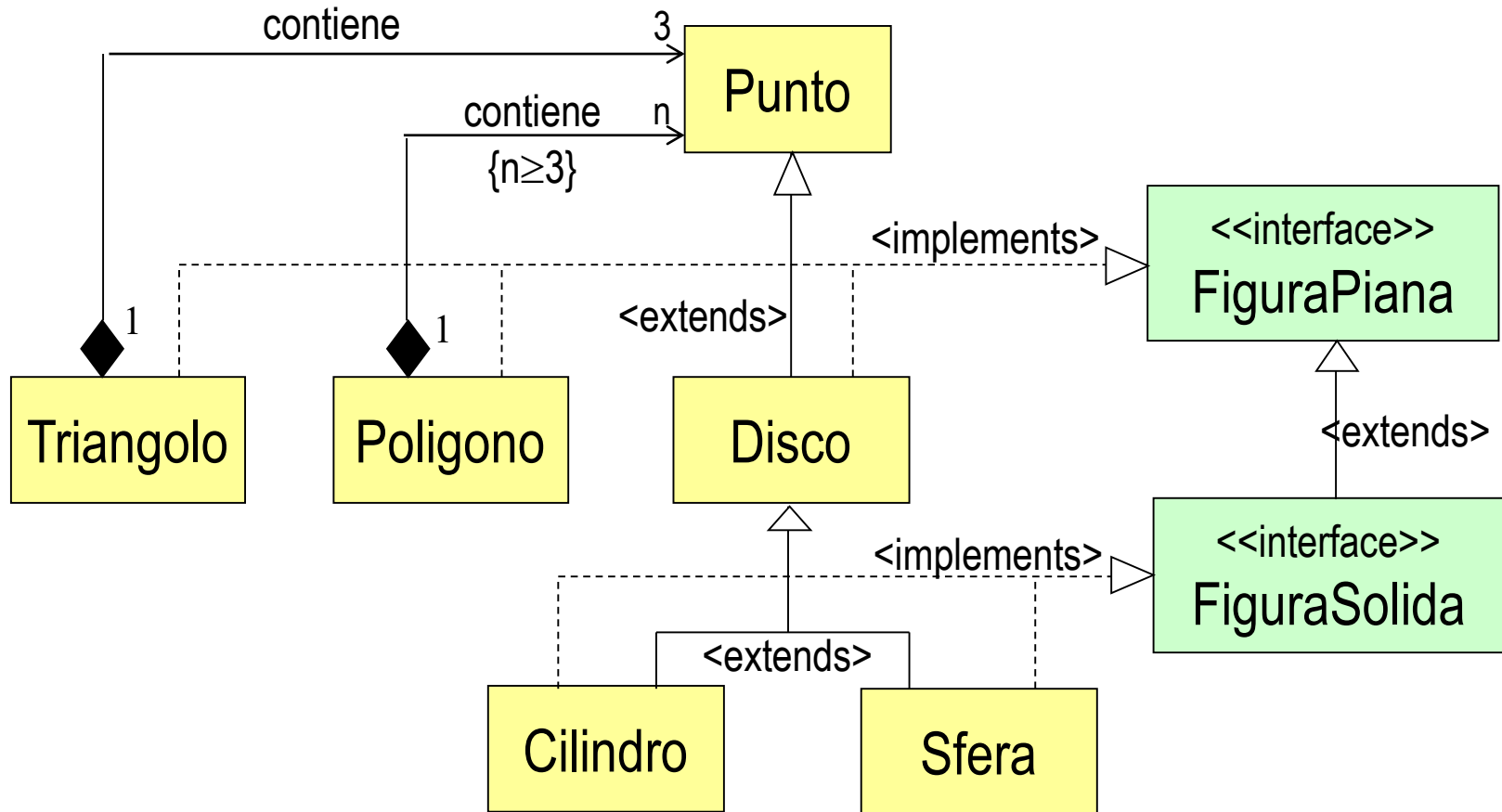
un'interfaccia può far parte di un package esplicito ed essere raccolta in un file; va compilata come le classi

```
package poo.geometria;  
public class Triangolo implements FiguraPiana{  
    ...  
    public double perimetro(){...}  
    public double area(){...}  
}//Triangolo
```

Così come per Disco, etc.

- Il discorso può continuare come segue. Da Disco si può derivare Sfera che è una figura solida
- A questo punto si potrebbe definire un'interfaccia `FiguraSolida` che `extends FiguraPiana` ed aggiunge metodi come `double areaLaterale()` e `double volume()` etc. Per semplicità, il centro di una sfera o il centro della base di un cilindro si suppongono appartenenti al piano x,y
- Ovviamente in `FiguraSolida` si ritrova il perimetro che non ha senso in una figura a 3 dimensioni. Implementando `FiguraSolida` in `Sfera` si potrebbe codificare `perimetro()` in modo da generare un errore (eccezione). Il metodo `area()`, invece, si intende che calcola l'area totale. Nel caso della sfera area laterale e area totale coincidono. In un Cilindro, altro possibile erede di `Cerchio`, le due aree sono distinte

Un altro scenario



L'interfaccia FiguraSolida

```
package poo.geometria;  
public interface FiguraSolida extends FiguraPiana{  
    double areaLaterale();  
    double volume();  
} //FiguraSolida
```

L'interfaccia FiguraSolida include i metodi di FiguraPiana.

```
public class Sfera extends Disco implements FiguraSolida{  
    ...  
    public double perimetro(){ throw new UnsupportedOperationException();}  
    public double area(){ return 4*Math.PI*raggio*raggio; }  
    public double areaLaterale(){ return 4*Math.PI*raggio*raggio; }  
    public double volume(){ return (4*Math.PI*raggio*raggio*raggio)/3; }  
} //Sfera
```



```
public class Cilindro extends Disco implements FiguraSolida{  
    private double altezza;  
    ...  
    public double perimetro(){ throw new UnsupportedOperationException();}  
    public double area(){ return areaLaterale()+2*raggio*raggio*Math.PI; }  
    public double areaLaterale(){ return 2*Math.PI*raggio*altezza; }  
    public double volume(){ return raggio*raggio*Math.PI*altezza; }  
} //Cilindro
```

Si è mostrata una gerarchia di classi ancorata sulla classe astratta Figura e una collezione di classi e interfacce basate sui punti.

Qual è l'approccio migliore ?

Non c'è una risposta assoluta: tutto dipende dalle esigenze della applicazione. Se sono sufficienti le dimensioni, si può optare per la prima soluzione, se servono i punti-vertici si può passare all'altra organizzazione.

Per "riconciliare" le due gerarchie di figure, è sufficiente che la classe astratta Figura implementi l'interfaccia FiguraPiana. Diventa così possibile introdurre un array di FiguraPiana in cui si possono collocare oggetti-figure dell'una o dell'altra gerarchia.

Come riconciliare le due gerarchie di figure piane

```
package poo.figure;  
import poo.geometria.FiguraPiana;  
  
public abstract class Figura implements FiguraPiana{  
    private double dimensione;  
    public Figura( double dim ){ this.dimensione=dim; }  
    protected getDimensione(){ return dimensione; }  
} //Figura
```

Siccome FiguraPiana definisce i metodi perimetro() e area(), essi si possono omettere nella scrittura della classe Figura, in quanto essendo non implementati in Figura, sono automaticamente assunti come metodi abstract.

Dichiarando ora un array come FiguraPiana[] f=new FiguraPiana[10]; diventa possibile inserire in f indifferentemente oggetti delle due gerarchie.

Stringhe di caratteri e classe String

- In Java le stringhe e gli array sono oggetti, dunque sono allocati nell'heap a seguito di una operazione new.

```
String s="casa"; //creazione implicita mediante aggregato di caratteri  
//crea un oggetto stringa col contenuto "casa"
```

- Mentre le classi degli array sono ignote al programmatore ma note solo al compilatore, le stringhe appartengono alla classe String di java.lang, che fornisce diversi metodi per la manipolazione.
- Gli oggetti String sono immutabili: una volta creata una stringa, essa non può più essere modificata. Tuttavia, data una variabile String s si può sostituire in s il riferimento ad un oggetto stringa con il riferimento ad un altro oggetto string.
- Gli oggetti String sono dotati del **confronto naturale** (compareTo()) o *lessicografico* (il primo carattere diverso da sinistra, se esiste, tra due stringhe s1 ed s2, stabilisce se s1 precede o segue o s2). Il confronto è case-sensitive. Es. "casa".compareTo("casaBlanca") è <0 (il carattere nullo dopo la seconda 'a' di casa, precede 'B') etc.
- Anche equals() è case-sensitive. Esiste equalsIgnoreCase() che verifica l'uguaglianza ignorando il caso dei caratteri.

Alcuni metodi di uso frequente

- `String s=new String();` //crea una stringa vuota, equivalente a: `""`
- `String s1=new String("Java is object oriented!");` //costruttore di copia
- I caratteri di `s1` sono indicati a partire da 0 a `s1.length()-1` (`s1.length()`=>24)
- *`char charAt(indice)`:*
 - `s1.charAt(3)`=>'a', `s1.charAt(6)`=>'s' etc.
- *`int indexOf(char x), int indexOf(char x, int a_partire_da_forward)`,*
- *`int lastIndexOf(char x), int lastIndexOf(char x, int a_partire_da_backward)`*
 - `int i=s1.indexOf(' ');` //ad i si assegna l'indice 4
 - `i=s1.indexOf(' ',i+1);` //ad i si assegna l'indice 7
- *`int indexOf(string), int indexOf(string, int)`*
 - `int j=s1.indexOf("is");` //j prende il valore 5
 - `j=s1.indexOf("ect",j+1);` //a j si assegna 11
- `String substring(int da, int a_escluso), String substring(int da) //fino a length()`
 - `String s2=s1.substring(0,4);` //s2 prende come valore "Java"
 - `int i=s1.lastIndexOf(' ');` //i punta al terzo spazio
 - `String s3=s1.substring(i);` //sottostringa da i a fine stringa: "oriented!"

- *String toUpperCase(), String toLowerCase()*
 - `s1=s1.toUpperCase();` //cambia la stringa in s1 con un'altra che è s1 tutto in maiuscolo
- *static String valueOf(tipo_di_base o Object)*
 - `String s3=String.valueOf(150);` //s3 prende la stringa "150", etc.
 - `s3=String.valueOf(new Razionale(3,5));` //s3 prende il toString del razionale
- *String trim()*
 - `String s=" a bad world Sir! ".trim();` // s prende la stringa "a bad world Sir!"
- *char[] toCharArray()*
- *String toString()*
- *int compareTo(String), int compareToIgnoreCase(String)*
 - `if("casa".compareTo("baco") < 0)` è false in quanto "casa" segue "baco"
- *boolean matches(String regex), String replaceAll(String regex, String other)*
 - saranno approfonditi più avanti nel corso, parlando delle espressioni regolari
- *String concat(String other)*
 - `s1=s1.concat(" Ok Watson?");` //s1 ora vale «Java is fantastic! Ok Watson?"
 - Equivalente a: `s1+=" Ok Watson?";`
- *static String format (stringa_formato, valore_tipo_primitivo o stringa)*
 - `String s=String.format("%1.2f", x)` dove x è un double di cui interessano 2 cifre frazionarie

Altri metodi di String

- Spesso si desidera che un certo pattern (sotto stringa) venga ripetuto un certo numero di volte in una stringa. Ciò si può ottenere (nelle versioni più recenti di Java) come segue:
- `String s=".".repeat(10);` //genera una stringa di 10 punti
- `String s1="A"+"AB".repeat(3)+"B";` //genera AABABABB
- `if(s1.equals("AABABABB")) ...`
- `String linea=sc.nextLine();`
- `String[] parole=linea.split("\\W+");`
- `for(int i=0; i<parole.length; ++i)` scrittura o altro di `parole[i];`

Esempio

- *Input*: si legge una linea contenente cognome e nome di una persona. Il cognome può essere preceduto da spazi. Il nome può essere seguito da spazi. Tra cognome e nome esiste almeno uno spazio
- *Output*: si deve scrivere l'iniziale del nome seguita da '.', quindi da uno spazio e quindi dal cognome
- Esempio di input: Gosling James INVIO
- Output corrispondente: J. Gosling

```
public class TestString {  
    public static void main( String[] args ){  
        Scanner sc=new Scanner(System.in);  
        System.out.println("Fornisci cognome e nome di una persona ");  
        String linea=sc.nextLine();  
        linea=linea.trim(); //elimina spazi iniziali e finali  
        int i=linea.indexOf(' ');  
        String cognome=linea.substring(0,i);  
        //salta spazi  
        while( i<=linea.length() && linea.charAt(i)==' ' ) i++;  
        String nome=linea.substring(i);  
        System.out.println(nome.charAt(0)+" "+cognome);  
    }  
}
```

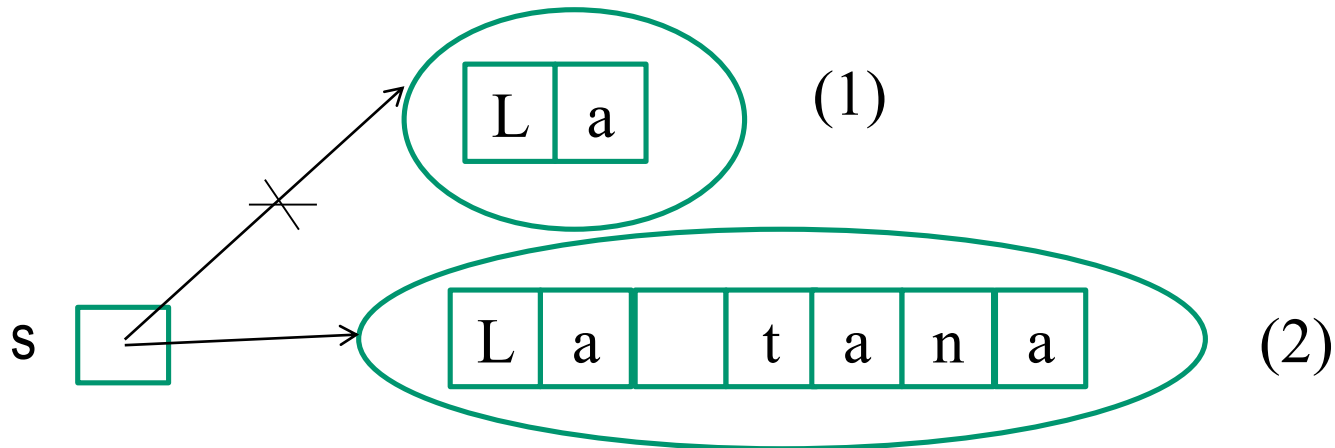

Uso di lastIndexOf()

```
public class TestString {  
    public static void main( String[] args ){  
        System.out.println("Fornisci cognome e nome di una persona ");  
        Scanner sc=new Scanner(System.in);  
        String linea=sc.nextLine();  
        linea=linea.trim();  
        int i=linea.indexOf(' ');  
        String cognome=linea.substring(0,i);  
        i=linea.lastIndexOf(' '); //trovato forward a partire da 0  
        //i=linea.lastIndexOf(' ',linea.length());  
        //fa la ricerca backward a partire dalla fine  
        String nome=linea.substring(i+1);  
        System.out.println(nome.charAt(0)+" "+cognome);  
    }  
}
```

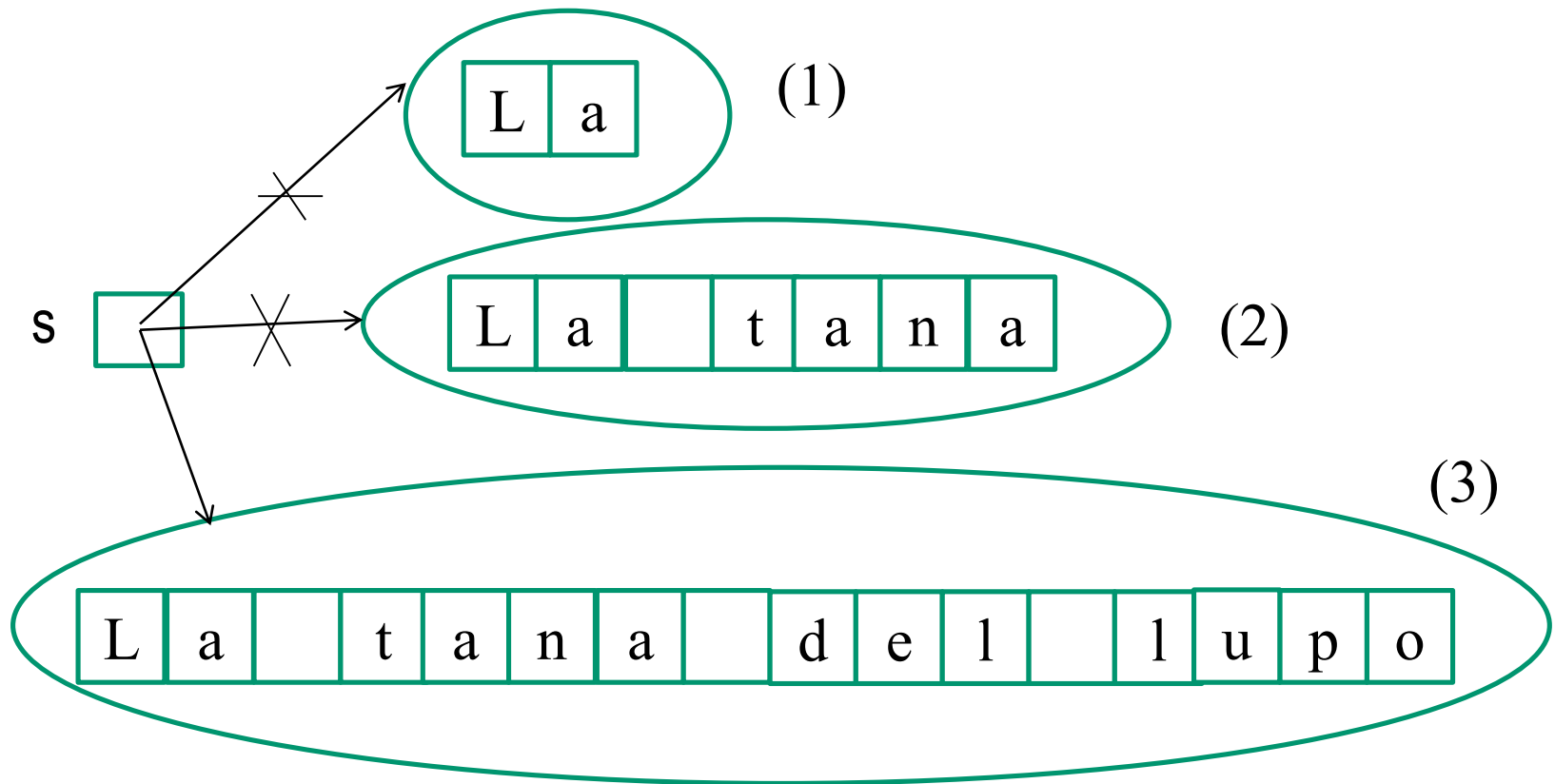
//main
//TestString

Proliferazione di stringhe garbage

- L'immutabilità delle stringhe comporta che molti metodi costruiscono una nuova stringa e la ritornano, così che un vecchio oggetto string venga buttato via e sia rimpiazzato da uno nuovo. Si osservi la seguente cascata di operazioni, alla luce del modello di memoria
- String s="La"; (1)
- s=s+" tana "; (2)
- s+=" del lupo"; (3)



- Al tempo (2) s non riferisce più l'oggetto stringa contenente "La" quanto il nuovo oggetto stringa nato dalla concatenazione (2) che contiene "La tana"
- Similmente al tempo della terza concatenazione s riferisce il nuovo oggetto String appena creato col valore "La tan
- Gli oggetti creati (1) e (2) sono diventati garbage e la loro memoria può essere raccolta dal garbage collector



Argomenti di un programma

- È noto che il metodo main riceve un array di String. I valori di questo array sono specificati, nella riga di comando che lancia l'esecuzione, subito dopo il nome del programma, separandoli da spazi:

c:\poo-java\java poo.string.TestArguments 10 8 12 -3 5 7 INVIO

- In questo caso viene chiesta l'esecuzione del main della classe TestArguments e viene costituito un array di stringhe es. args con i valori che seguono il nome del programma: args[0]="10" args[1]="8" args[2]="12" ...
- Il programma può quindi consultare l'array di stringhe ottenuto e prelevarvi i valori (String) trasmessi
- Nel programma che segue si costruisce un array di interi i cui valori sono forniti come argomenti del programma, si ordina l'array e si mostra il contenuto
- In generale, l'uso di argomenti in un programma ha senso quando sono in gioco pochi valori. Diversamente, occorre effettuare le usuali operazioni di input es. attraverso uno Scanner

```
package poo.string;
import java.util.*;
public class TestArguments {
    public static void main( String []args ){
        if( args.length==0 ){
            System.out.println("Argomenti assenti"); System.exit(-1);
        }
        int []v=new int[args.length];
        for( int i=0; i<v.length; i++ ){
            v[i]=Integer.parseInt( args[i] );
        }
        System.out.println( Arrays.toString( v ) ); //v iniziale
        poo.util.Array.bubbleSort(v);
        System.out.println( Arrays.toString( v ) ); //v ordinato
    } //main
} //TestArguments
```

- Si è detto che gli argomenti di un programma sono stringhe poste sulla linea di comando, separate da spazi
- Nel caso in cui certi spazi devono far parte di una stessa stringa argomento, è sufficiente racchiudere il tutto tra “ e “:
c:\poo-java\java poo.string.Prog “A e B” “C e D” INVIO
- In questo caso vengono trasmessi al main di Prog **due** argomenti: “A e B” e “C e D”
- La possibilità di passare argomenti ad un programma dall'interno di eclipse si basa sulla costruzione di una “configurazione di run” (si sceglie l'opzione Run As->Run Configurations ...) in cui si specifica il progetto, il package e la classe col main da lanciare, e gli argomenti da fornire (nel tab Arguments->Program arguments:). Eventuali opzioni da passare alla Java Virtual Machine (JVM) si possono inserire nel tab VM arguments.

StringBuffer o StringBuilder di java.lang

- Sono due **classi di stringhe mutabili** che utilizzano un array sottostante scalabile dinamicamente. In assenza di ambiente multithread (si veda più avanti nel corso) è preferibile (in quanto più efficiente) utilizzare uno StringBuilder.
- Rispondono a stessi metodi quali:
 - *char charAt(indice)*
 - *void setCharAt(indice i, char x)* //cambia il carattere alla posizione i con quello x
 - *StringBuilder append(tipo_di_base o oggetto)*
 - *int length(), void setLength(len)* // fissa la lunghezza a len; es. se len è 0, "resetta"
 - *String toString()*
- Il più interessante è *append()*. Esso riceve un valore di un tipo primitivo (int, char, boolean, double etc.) o un oggetto, lo converte a stringa di caratteri e concatena questi caratteri al contenuto attuale dello string builder (array) se necessario espandendone prima la dimensione
- L'uso oculato di uno StringBuilder suggerisce di dimensionarlo inizialmente ad una capacità opportuna per evitare la riallocazione dell'array sottostante che avrebbe effetti simili a quelli visti con le stringhe garbage
- Il costruttore di default *StringBuilder()* crea uno string builder con capacità iniziale di 16 caratteri
- Il costruttore *StringBuilder(capacity)* crea uno string builder di una certa capacità (iniziale)
- Il metodo accessore *int capacity()* consente di ispezionare il valore corrente della capacità

- Si considera una classe C che ammette un array **a** di double ed un intero **size** (dimensione effettiva di riempimento dell'array) come variabili di istanza
- Si vuole scrivere il metodo toString() di C. Non esiste una soluzione "perfetta"
- Ogni volta che si invoca il toString() occorre creare una nuova String che contenga lo stato dell'oggetto this sotto veste di stringa
- Si mostra una soluzione "classica" basata sulla concatenazione di oggetti String, ed una basata su uno StringBuilder dimensionato appropriatamente
- Gli elementi di **a** vengono separati da ',' e avviluppati tra una [e una]
- L'uso di uno StringBuilder *può* essere favorevole dal punto di vista temporale


```
public class C{
    private double []a; //creato nel costruttore
    private int size; //dimensione effettiva di a
    ...
    public String toString(){ //soluzione basata su concatenazione di String
        String s="[";
        for( int i=0; i<size; i++ ){
            s+=String.format( "%1.2f",a[i] );
            if( i<size-1 ) s+=", "; //,+spazio
        }
        s+="]";
        return s;
    }//toString
}//C
```

```
public class C{
    private double []a; //creato nel costruttore
    private int size; //dimensione effettiva di a
    ...
    public String toString(){
        StringBuilder sb=new StringBuilder(500); //esempio
        sb.append('[');
        for( int i=0; i<size; i++ ){
            sb.append( String.format( "%1.2f",a[i] ) );
            if( i<size-1 ) sb.append(", "); //, e spazio
        }
        sb.append(']');
        return sb.toString();
    } //toString
} //C
```

java.util.StringTokenizer (legacy, non agganciata alle regex)

- Spesso si ha una stringa e si desidera frammentarla nei suoi costituenti (token)
- L'esempio classico è una linea di testo costituita da parole alfanumeriche, separate una dall'altra da spazi bianchi o segni di punteggiatura
- Ovviamente, lavorando con i metodi di String si potrebbe agevolmente ottenere la scomposizione. Tuttavia, la classe StringTokenizer di java.util permette una soluzione più intuitiva. I costruttori più interessanti sono due:
 - *StringTokenizer(String string, String delimitatori)*
 - *StringTokenizer(String string, String delimitatori, boolean ritornoDelimitatori)*
- Nel primo caso i delimitatori consentono di individuare il prossimo token, ma per il resto sono saltati automaticamente
- Nel secondo caso, i delimitatori non solo sono utilizzati per ottenere i token, ma essi stessi sono ottenibili come token
- Il metodo che ritorna il prossimo token da uno string tokenizer è
 - *String nextToken()*
- Il metodo che controlla se esistono altri token nella stringa è:
 - *boolean hasMoreTokens()*

Tokenizzazione di una linea in input

```
import java.util.*;

...

Scanner sc=new Scanner( System.in );

String linea=sc.nextLine();

StringTokenizer st=new StringTokenizer( linea, " ,;:." ); //esempio

while( st.hasMoreTokens() ){
    String tk=st.nextToken();
    System.out.println("Token ottenuto: "+tk );
}
```

Tokenizzazione mediante Scanner

- Oltre che mediante uno StringTokenizer, la suddivisione in token di una stringa-linea può essere ottenuta tramite la classe Scanner e metodi associati. Nell'ipotesi che i delimitatori dei token siano caratteri non alfabetici, si può operare come segue

String linea=...

Scanner sl=new Scanner(linea); //scanner aperto sulla stringa linea

sl.useDelimiter("[^A-Za-z]+"); //fissa i delimitatori con una espressione regolare

while(sl.hasNext()){

 String tk=sl.next();

processa tk

}

- Mentre per l'approfondimento delle espressioni regolari si rimanda a più avanti nel corso, si nota la flessibilità e sinteticità nella definizione dei delimitatori. I metodi di Scanner sono utilizzati per ottenere in sequenza i token, saltando i delimitatori
- Anziché manipolare token String con la coppia hasNext()/next() si possono scandire interi o double con hasNextInt()/nextInt(), hasNextDouble()/nextDouble()
- La richiesta di un token che non esiste (es. hasNext() è false), solleva l'eccezione NoSuchElementException.

Valutazione di un'espressione aritmetica intera

- L'input di un programma è costituito da un'espressione intera con gli usuali operatori binari $+$, $-$, $*$ e $/$
- Per semplicità, non sono ammessi spazi e la valutazione procede strettamente da sinistra a destra, senza considerare le precedenze matematiche degli operatori
- Ad esempio: $30+40*2$ si valuta a 140 e non a 110
- Gli operandi sono interi senza segno
- Il programma ottiene un'espressione (da riga di comando o da tastiera), la tokenizza nei suoi costituenti (numeri e segni di operazioni), la valuta e quindi scrive il risultato su output. Gli operatori fungono *anche* da separatori degli operandi; essi vanno espressamente restituiti al programma per la valutazione del risultato

```
package poo.string;  
import java.util.*;
```

```
public class ValutatoreEspressione {  
    public static void main( String []args ){  
        String espr=null;  
        if( args.length==1 ){  
            espr=args[0];  
        }  
        else{  
            Scanner sc=new Scanner(System.in);  
            System.out.print("Espr>");  
            espr=sc.nextLine();  
        }  
    }  
}
```

```
StringTokenizer st=new StringTokenizer(espr,"+-*/",true);
int ris=Integer.parseInt(st.nextToken());
while( st.hasMoreTokens() ){
    char op=st.nextToken().charAt(0);//ottiene l'operatore
    int num=Integer.parseInt( st.nextToken() );
    switch(op){
        case '+': ris=ris+num; break;
        case '-': ris=ris-num; break;
        case '*': ris=ris*num; break;
        default: ris=ris/num;
    }
}
System.out.println(espr+"="+ris);
} //main
} // ValutatoreEspressione
```


Generalizzazione del valutatore

- Al fine di recuperare le precedenze degli operatori della matematica, si decide di accettare anche che sotto espressioni possano essere racchiuse tra (e)
- Una (sotto) espressione in parentesi si valuta sempre prima
- Si decide di sviluppare la classe in versione instanziabile e basata su due metodi di valutazione: `valutaOperando(...)` e `valutaEspressione(...)`. Quando si incontra una (si invoca ricorsivamente `valutaEspressione(...)`.
- Vediamo i dettagli.

```
package poo.string;
```

```
import java.util.StringTokenizer;
```

```
public class ValutatoreEspressione {
```

```
    private String expr;
```

```
    public ValutatoreEspressione( String expr ) {
```

```
        //validazione expr mediante regex - rimandata
```

```
        this.expr=expr;
```

```
    }
```

```
    public int valuta() {
```

```
        StringTokenizer st=new StringTokenizer(expr,"+-*/()",true);
```

```
        return valutaEspressione(st);
```

```
    } //valuta
```

```

private int valutaOperando( StringTokenizer st ) {
    String tok=st.nextToken();
    if( tok.charAt(0)=='(' ) return valutaEspressione(st);
    return Integer.parseInt(tok);
} //valutaOperando

private int valutaEspressione( StringTokenizer st ) {
    int ris=valutaOperando(st); //risultato parziale
    while( st.hasMoreTokens() ) {
        char op=st.nextToken().charAt(0);
        if( op=='(' ) return ris;
        int opnd=valutaOperando(st);
        switch( op ) {
            case '+': ris=ris+opnd; break;
            case '-': ris=ris-opnd; break;
            case '*': ris=ris*opnd; break;
            case '/': ris=ris/opnd; break;
            default : throw new RuntimeException(op+" inatteso");
        }
    }
    return ris;
} //valutaEspressione

```

```
public static void main( String[] args ) {  
    //espressione attesa a riga di comando  
    if( args.length==0 || args.length>1 ) {  
        System.out.println("Attesa espressione.");  
        System.exit(-1);  
    }  
    ValutatoreEspressione v=  
        new ValutatoreEspressione( args[0] );  
    System.out.println(args[0]+"="+v.valuta());  
} //main  
  
} //ValutatoreEspressione
```

Si può far partire ValutatoreEspressione o a riga di comando o dall'interno di Eclipse.

Concetti sulle Eccezioni

- Un metodo Java o **termina normalmente** producendo il suo risultato (eventualmente void) o **termina in modo eccezionale** restituendo al chiamante una **eccezione** per segnalare che nelle condizioni in cui il metodo è stato invocato (con i valori specificati dei parametri) esso non è grado di generare un risultato valido
- Si parla di **servizio normale** e **servizio eccezionale** prodotto da un metodo a seconda che esso termini normalmente o in veste eccezionale
- In Java le **eccezioni** sono **oggetti**, ossia istanze di classi particolari
- E' possibile **sollevare, catturare e gestire un'eccezione** (*exception handling*) generata da un metodo, dopo di che, eventualmente, il metodo potrebbe essere ri-invocato o comunque la computazione del programma potrebbe continuare verso la sua conclusione
- Un caso particolare di eccezioni sono gli Error, che modellano situazioni serie di errori che un programma non si aspetta di poter gestire (catturare e ricoverare) (es. OutOfMemory error)

Un esempio

```
package poo.razionali;
public class Razionale implements Comparable{
    private int numeratore, denominatore;
    public Razionale( int num, int den ) throws DenominatoreNullo{
        if( den==0 ) throw new DenominatoreNullo();
        if( num!=0 ){ //riduzione ai minimi termini
            int n=Math.abs( num ), d=Math.abs( den );
            int cd=Mat.mcd( n, d );
            num=num/cd; den=den/cd;
        }
        if( den<0 ){ num *= -1; den *= -1; }
        this.numeratore=num;
        this.denominatore=den;
    } //costruttore
    ...
} //Razionale
```

La classe DenominatoreNullo

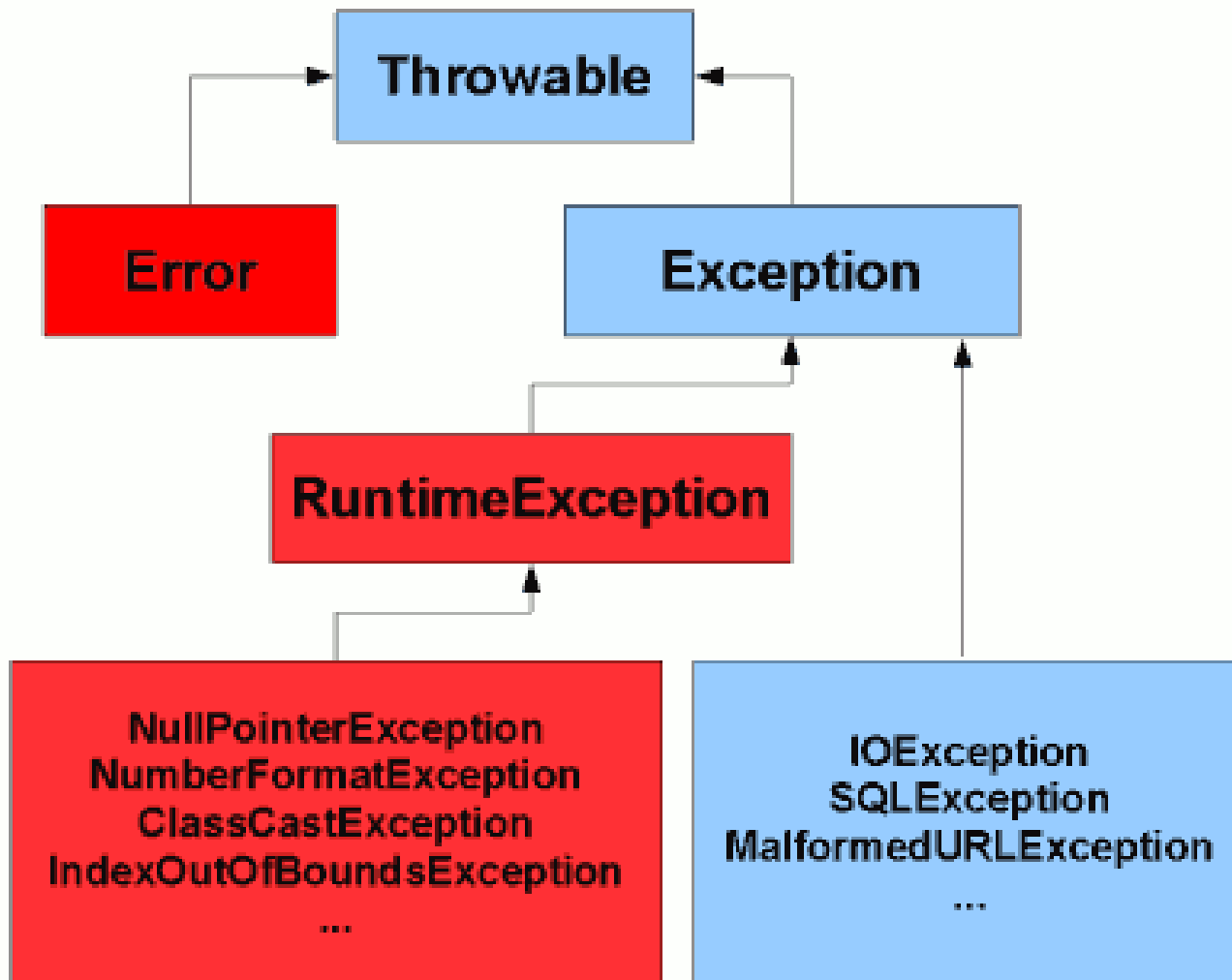
```
package poo.razionali;  
public class DenominatoreNullo extends Exception{  
    public DenominatoreNullo(){  
    public DenominatoreNullo( String msg ){  
        super( msg );  
    }  
} //DenominatoreNullo
```

Esempio di cattura e gestione dell'eccezione

```
//in un main
Scanner sc=new Scanner( System.in );
Razionale []v=new Razionale[10];
//caricamento di v
int i=0, n=0 /*fittizio*/, d=0 /*fittizio*/;
loop: while( i<v.length ){
    System.out.print("numeratore= "); n=sc.nextInt();
    System.out.print("denominatore= "); d=sc.nextInt();
    try{
        v[i]=new Razionale(n, d); //può sollevare l'eccezione DenominatoreNullo
    }catch(DenominatoreNullo e){//cattura e gestione eccezione
        System.out.println("Denominatore nullo!");
        System.out.println("Ridare il razionale"); continue loop;
    }
    i++; //conta questo razionale
} //while
```

- L'istruzione **continue** loop consente di continuare immediatamente il ciclo etichettato (il while) saltando tutte le istruzioni che seguono (nell'esempio la i++) sino alla fine del corpo del ciclo. Dunque, in caso di eccezione DenominatoreNullo, la i non viene incrementata e si torna a leggere una nuova coppia <numeratore, denominatore> per la stessa posizione i dell'array v

Gerarchia classi di eccezioni



Eccezioni checked e unchecked

Le eccezioni sono oggetti di classi che possono essere programmate o come eredi di **Exception** o come eredi di **RuntimeException**

Le eccezioni che derivano da `Exception` sono dette *checked* (o controllate), quelle che derivano da `RuntimeException` *unchecked* (o non controllate)

Quando un'eccezione è del tipo checked allora il programmatore non può ignorarla: è necessario aggiungere la clausola **throws** NomeEccezione alla intestazione del metodo e, presso il chiamante del metodo, occorre utilizzare il blocco **try-catch** o propagare l'eccezione (si veda più avanti)

Se l'eccezione è di tipo unchecked allora non esiste alcun obbligo sul programmatore. Tuttavia, se l'errore si verifica, il programma termina con una segnalazione diagnostica.

Come scegliere ?

Generalmente

- si usano eccezioni del tipo **RuntimeException** in tutti quei casi in cui è il programma al suo interno che può, generare il malfunzionamento o guasto (fault), es. divisione per 0, tentare di seguire un riferimento null etc. In queste circostanze, con un test (if) è possibile evitare l'insorgere del guasto;
- si usano eccezioni del tipo **Exception** se il fault non dipende dal programma ma piuttosto da condizioni esterne al programma, ad es. situazioni di errore durante operazioni di I/O, di accesso a file, di accesso alla rete di comunicazione etc

- Le classi di eccezioni predefinite **ArrayIndexOutOfBoundsException**, **NullPointerException**, **IllegalArgumentException**, **ClassCastException** etc.

sono eredi di **RuntimeException**

- L'aver programmato DenominatoreNullo come classe erede di Exception è stata una scelta di esempio, per scopi dimostrativi
- Siccome il fault è perfettamente evitabile dall'utente, si poteva utilizzare la strada unchecked. Vediamo come

```
package poo.razionali;
```

```
public class DenominatoreNullo extends RuntimeException{
```

```
    public DenominatoreNullo(){}
```


```
    public DenominatoreNullo( String msg ){
```

```
        super( msg );
```

```
    }
```

```
}//DenominatoreNullo
```

```
package poo.razionali;
public class Razionale implements Comparable{
    private int numeratore, denominatore;
    public Razionale( int num, int den ) {
        if( den==0 ) throw new DenominatoreNullo();
        //throw new RuntimeException("DenominatoreNullo");
        ...
    }//costruttore
    ...
} //Razionale
```



quest'alternativa
evita del tutto la classe
DenominatoreNullo

Nuovo main

```
Scanner sc=new Scanner( System.in );
Razionale []v=new Razionale[10];
//caricamento di v
int i=0, n=0 /*fittizio*/, d=0 /*fittizio*/;
loop: while( i<v.length ){
    System.out.print("numeratore= "); n=sc.nextInt();
    System.out.print("denominatore= "); d=sc.nextInt();
    if( d==0 ){
        System.out.println("Denominatore nullo!");
        System.out.println("Ridare il razionale"); continue loop;
    }
    v[i]=new Razionale(n, d); //il costruttore non può più sollevare
    l'eccezione
    i++; //conta questo razionale
} //while
```

- Nel nuovo main l'eccezione DenominatoreNullo è **evitata** con il test preliminare
- Naturalmente, se non ci si caute (svista del programmatore che “assume” un denominatore sempre non nullo e dunque non esegue il test `if(d==0) ...`), allora se l'eccezione `RuntimeException` viene sollevata, essa causa la terminazione anticipata del programma
- Si nota esplicitamente che anche un'eccezione di tipo `RuntimeException` può essere catturata e gestita con un blocco **try-catch**

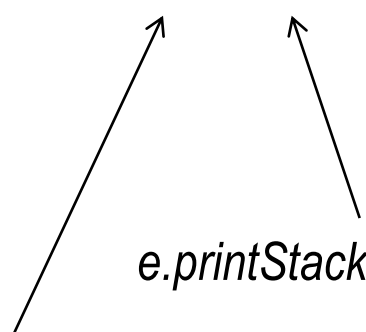
Vincoli sulle eccezioni checked

- Se un metodo *m2* di un oggetto *o2*, invocato dall'interno di un metodo *m1* di un oggetto *o1*, può sollevare un'eccezione checked allora:
 - o si avviluppa l'invocazione di *m2* in un blocco try-catch
 - o si dichiara che il metodo *m1* può a sua volta propagare l'eccezione
- La propagazione di un'eccezione può essere esplicita anche dall'interno di un blocco try-catch con l'istruzione **throw**
- Di seguito si schematizzano i due possibili comportamenti

Eccezioni checked

Strada 1: *gestione*

```
tipo_ritorno m1(params){  
    ...  
    try{  
        o2.m2(parametri);  
    }catch(TipoEccezione e){  
        azioni "correttive"  
    }  
    ...  
} //m1
```



e.printStackTrace()

Dopo la gestione dell'eccezione, m2 potrebbe essere re-invocato

Strada 2: *propagazione*

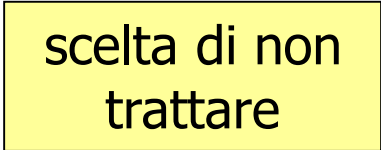
```
tipo_ritorno m1(params) throws TipoEccezione  
{  
    ...  
    o2.m2(parametri);  
    ...  
} //m1
```

In questo caso, se m2 si conclude in modo eccezionale, non essendoci gestione in m1, l'eccezione è tacitamente propagata al chiamante di m1 etc

La clausola **throws** può specificare una lista di nomi di classi di eccezioni, separate da ',':
throws CE1, CE2, ...

Se DenominatoreNullo estende Exception, si può avere

```
public static void main( String []args ) throws DenominatoreNullo{
    Razionale []v=new Razionale[10];
    //caricamento di v
    int i=0;
    loop: while( i<v.length ){
        int n=Console.readInt("numeratore= ");
        int d=Console.readInt(" denominatore= ");
        if( d==0 ){
            System.out.println("Denominatore nullo!");
            System.out.println("Ridare il razionale");
            continue loop;
        }
        v[i]=new Razionale(n, d); //non può sollevare l'eccezione checked
        i++; //conta questo razionale
    }//while
}//main
```



scelta di non
trattare

Eccezioni unchecked

- Non esiste l'obbligo di dichiarare, nella testata dei metodi, che essi sollevano eccezioni unchecked
- Non esiste l'obbligo di catturare e gestire le eccezioni unchecked mediante blocchi try-catch
- Non esiste l'obbligo per un metodo che non cattura un'eccezione unchecked, di dichiarare che esso ripropaga l'eccezione al chiamante
- Naturalmente: l'assenza di obblighi non è “tranquillità”: se accade un'eccezione unchecked non gestita, il programma termina in eccezione.

Il blocco try-catch

- Può includere più istruzioni ciascuna delle quali può sollevare una o più eccezioni
- Può ammettere più clausole **catch** per catturare e gestire ciascuna possibile eccezione generata all'interno del corpo try-catch
- Può ammettere una (opzionale) clausola **finally** che, se presente, specifica codice che viene eseguito *dopo* la gestione di una qualsiasi eccezione e comunque sempre all'uscita del blocco try (anche senza che si siano verificate eccezioni)

```
try{  
    istruzione_i1;  
    istruzione_i2;  
    ...  
}catch(ClasseEccezioneA eA){ gestione eA }  
catch(ClasseEccezioneB eB){ gestione eB }  
...  
[finally{ azioni finali }]
```

Attenzione: L'uso in una clausola catch di una classe base di eccezioni (es. **Exception**) consente di catturare più di un tipo di eccezioni. Ovviamente, in questi casi, *l'exception handler* può avvalersi di **instanceof** per scoprire il tipo specifico dell'oggetto eccezione e intraprendere le azioni correttive corrispondenti

Il blocco try-finally

- È utile, indipendentemente dalla gestione di eccezioni, per eseguire delle istruzioni e quindi (in ogni caso) concludere con delle azioni finali (es. chiusura di un file, di una connessione di rete, apertura di un lucchetto di sincronizzazione etc.)

```
try{  
    istr1;  
    istr2;  
    ...  
}finally{  
    azioni finali  
}
```

Anche la chiusura
di uno scanner (file etc) potrebbe
sfruttare un blocco try-finally

- Si nota che le azioni del corpo finally {} sono eseguite comunque si esca dal blocco try, anche tramite una return

Flusso del controllo

- Si consideri la **catena dinamica**:

`o1.m1(...) -> o2.m2(...) -> o3.m3(...)`

- In assenza di eccezioni, m1 si sospende in attesa che m2 finisca il suo compito, m2 a sua volta si sospende in attesa che m3 finisca il suo compito etc. Quando m3 termina, riprende m2. Quando m2 termina riprende m1. Tutto ciò è **normale**...
- La relazione chiamante->chiamato è importante non solo per la restituzione dei risultati ma anche in presenza delle eccezioni
- Se m3 anzichè fornire il servizio ordinario genera un'eccezione e1, allora tale eccezione (che sostituisce il normale risultato) viene riportata ad m2 nel punto dove m2 chiama m3. Se in questo punto è presente un exception handler, allora è possibile che m2 risolva il problema e magari re-invochi m3 con nuovi parametri (senza che m1 ne sappia niente). Ma se in m2 non è presente alcun gestore di eccezione, allora anche m2 termina in errore e l'eccezione viene portata ad m1 nel punto dove m1 chiama m2
- Se m1 non tratta l'eccezione allora essa viene propagata ancora a monte, al suo chiamante, sino a raggiungere eventualmente il main che se propaga ancora allora fa sì che il programma termini **definitivamente** in errore

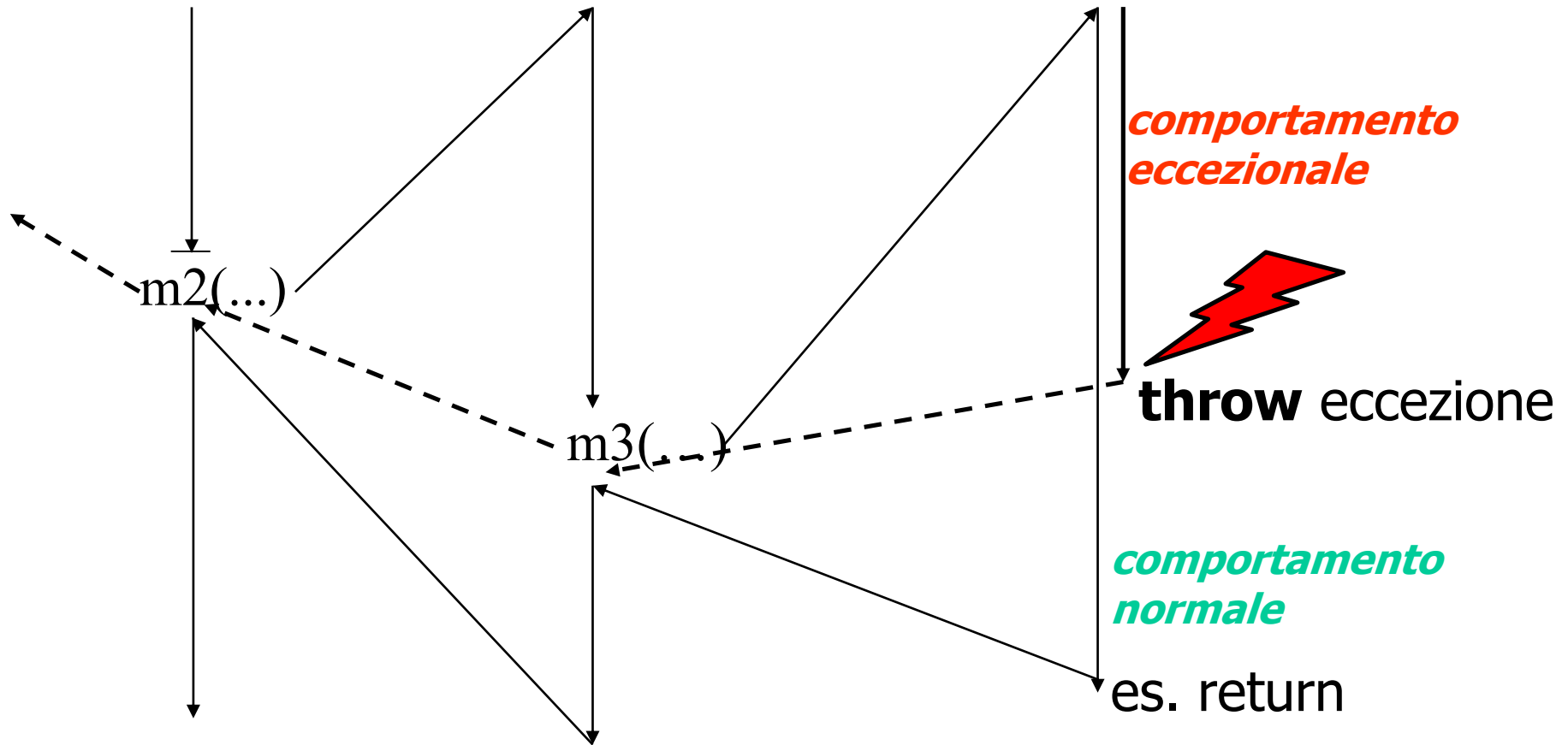
Flusso del controllo senza gestione delle eccezioni

forward →

m1(...)

m2(...)

m3(...)



backward ←