

# LINGUAGGI DI PROGRAMMAZIONE MODULO 1 TESTO PER ESAME

## NOTE

### ■ CONVENZIONI

```
public class Point{  
    //variabili  
    //costruttore e metodi  
    public Point(){  
    }  
}
```

- Nomi delle Classi in maiuscolo
- nomi delle istanze in minuscolo
- Le COSTANTI vanno dichiarate static final e devono essere totalmente MAIUSCOLE.
- Metodi con un nome composto da più parole inizia con la minuscola e parole successive devono iniziare con la Maiuscola
- Utilizzare un package "univoco" <- it.unitn.latemar.MOD1.Ronchetti.YIN.202453
- Non scrivere tutto nel Main/Start

### ■ PUNTATORI

```
Point punto = new Point();
```

- punto è una sorta di puntatore ad una area di memoria. Tipo puntatore ma senza algebra dei puntatori.
- Non è necessario free o malloc. Java possiede un garbage collector. Chiamata esplicita tramite `System.gc()`;
- Manca il distruttore `~Punto()`. In caso di necessità di eseguire operazioni prima della distruzione si usi il metodo `finalize()`

### ■ ARRAY

```
int[] i=new int[10];  
float[][] f=new float[10][10];  
Persona[] p=new Persona[30]
```

- La dichiarazione NON alloca lo spazio. Necessario il comando `New`. Array sono oggetti

### ■ ASSERT

```
assert(marker>0) : "Estrazione da un pila vuota!";
```

equivale a

```
if(marker==0){  
    System.out.println("Estrazione da una pila vuota!");  
    System.exit(1);  
}
```

### ■ EXCEPTION HANDLING

```
try{ }catch(Exception ex){ }finally{ }
```

- The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs

### ■ SYSTEM

```
System.exit(1);  
System.currentTimeMillis();  
System.gc(); System.runFinalization(); <- chiamata del gc "forse"
```

### ■ RANDOM

- Generatore di numeri casuali

```
public static Random rand = new Random(System.currentTimeMillis());
```

### ■ MATH

- Vedi API

```
normalizzazione permette di tradurre un valore y compreso tra ymin e ymax in un intero compreso tra 0 e nmax  
private int discretizza (double y, double ymin, double ymax, double nmax){  
    return (int)((nmax-1)*(1-(y-ymin)/(ymax-ymin)));  
}
```



- Metodo per “normalizzare”.

## ■ STRINGHE

- Verificare che una stringa contenga un certo carattere o sottostringa

```
"012345678".contains(key)
```

```
stringa.contains(carattere)
```

## ■ INPUT NO GRAFICA. PER GRAFICA VEDI TEXTDIALOG

```
import java.util.Scanner;
Scanner scanner = new Scanner(System.in); System.out.println("dimmi qualcosa");
String inputString = scanner.nextLine();
System.out.println(inputString);
```

- Esempio di input perfetto e safe con try catch finally

```
Scanner scanner = new Scanner(System.in);
String inputString;      int z;
boolean failure=true;
do {
    try {
        System.out.println("dammi un numero");
        inputString= scanner.nextLine();      z=Integer.parseInt(inputString);      failure=false;
    } catch (NumberFormatException ex) {      failure=true;
    } finally{}
} while (failure);
```

## ■ EREDITARIETÀ

```
public classe A{ }
public class B extends A{ }
```

- B eredita tutto da A.
- Solo ereditarietà singola. NON esiste ereditarietà multipla
- Tutte le classi EREDITANO Object implicitamente
  - public boolean equals(Object);
  - protected void finalize(); <- Insieme di azioni da eseguire prima della distruzione dell'oggetto
  - public String toString(); <- Viene chiamato automaticamente se l'oggetto ha bisogno di essere convertito in stringa
  - public int compareTo(Object)
  - public int hashCode();
  - protected Object clone(); <- Shallow Copy
- *super* -> punta alla sovraclassa.
- *this*-> punta all'oggetto corrente.

■ **Overloading** -> metodi stesso nome, argomenti diversi. Firma : nome del metodo, tipi degli argomenti. NON tipo di ritorno del metodo

■ **Overriding**-> classe figlia ridefinisce un metodo della classe padre. Applicazione se e solo se i metodi hanno la stessa firma.

```
Object o = new AutomobileElettrica();
Automobile a = o; // errato, Object non è un sottotipo di
Automobile Automobile a = (Automobile) o; // corretto (casting)
```

■ @Override è una buona cosa. Netbeans capisce che c'è un corso un @Override

- Tutte le classi ereditano il costruttore di Object. Se il costruttore NON è definito, ne viene usato uno standard senza argomenti. Se viene definito almeno un costruttore(anche con attributi) NON viene usato il costruttore standard

## ■ POLIMORFISMO E CASTING

- **Polimorfismo** -> capacità di un elemento sintattico di riferirsi ad elementi di diverso tipo
- Conversione forzata tra tipi
- Upcasting sempre possibile. Può avvenire implicitamente
  - **Liskov**: Se B è un sottotipo di A, le variabili di tipo A possono essere sostituite da B senza alterare la funzionalità. Tutto quello che ha A, lo ha anche B(sottotipo)
- Downcasting non sempre possibile. Necessaria dichiarazione esplicita

## ■ BINDING

- **Binding dinamico(Lazy binding)** -> tipo scelto in base al tipo dinamico(run-time)
- **Binding statico** -> tipo scelto in base al tipo statico(compilazione)
- **Regola 1:** Il compilatore determina la firma del metodo da eseguire SEMPRE basandosi sul tipo statico
- **Regola 2:** SE E SOLO SE c'è *@Overriding* la specifica implementazione del metodo la cui firma è già stata determinata tramite Regola 1 viene determinata a run-time in base al tipo dinamico

## ■ ABSTRACT E INTERFACCE

- Classe *abstract* non istanziabile. *abstract* se esiste almeno un metodo *abstract*.
- Vi è un metodo che provvede una firma ma non è implementato(*abstract*). Per l'utilizzo è necessario un *@Override* di tutti i metodi *abstract* da una classe figlia.

- **Interfacce** -> sono delle classi *abstract*. Non sono istanziabili. Provvedono a fornire una firma per tutte le classi figlie. Utilizzo tramite *implements*. Hanno ereditarietà multipla. Possono essere ereditate ed ereditare in maniera multipla

## ■ FINAL E STATIC

### • final

- *final class C{};* <- C cannot be inherited by any means
- *final int c;* <- c cannot be modified after creation

```
public final static Random random = new Random(System.currentTimeMillis());
```

### • static

- Si riferisce ad attributi di Classe e non di Oggetto. Memoria condivisa tra istanze dello stesso oggetto(NON USARE COME VARIABILE GLOBALE)
- Possibile chiamata tramite Classe.nome senza istanza
- Attributi static vanno inizializzati immediatamente.

## ■ INFORMATION HIDING IN JAVA

|                  | visibilità |         |                 |       |
|------------------|------------|---------|-----------------|-------|
| modificatore     | classe     | package | sottoclass<br>e | mondo |
| <b>private</b>   | Y          | N       | N               | N     |
| <b>"package"</b> | Y          | Y       | N               | N     |
| <b>protected</b> | Y          | Y       | Y               | N     |
| <b>public</b>    | Y          | Y       | Y               | Y     |

|            |      | classi (implementazioni) |                    |                  |                |
|------------|------|--------------------------|--------------------|------------------|----------------|
|            |      | hash<br>table            | resizable<br>array | balanced<br>tree | linked<br>list |
| interfacce | Set  | HashSet                  |                    | TreeSet          |                |
|            | List |                          | ArrayList          |                  | LinkedList     |
|            | Map  | HashMap                  |                    | TreeMap          |                |

## ■ COLLECTIONS -> VEDI OBSERVABLE LIST PER GESTIONE DI EVENTI QUANDO LA COLLECTION VIENE MODIFICATA

```
import java.util.*;
ArrayList collezione = new ArrayList<Integer>(); // Ricorda il cast
List<Integer> lista = new List<Integer>(); // Non è più necessario il cast
```

- Una Collection è un oggetto che raggruppa elementi multipli in una singola entità
- Contiene :
  - Interfacce
  - Implementazioni
  - Algoritmi
- **Metodi Base**
  - *int size();*
  - *boolean isEmpty();*
  - *boolean contains(Object element);*
  - *boolean add(Object element);* <- ritorna true se la collection è cambiata
  - *boolean remove(Object element);* <- ritorna true se la collection è cambiata
  - *Iterator iterator();*
- **Bulk Operations(più elementi)**
  - *boolean containsAll(Collection c);*

- boolean addAll(Collection c);
- boolean removeAll(Collection c);
- boolean retainAll(Collection c);
- void clear();

- **VISITA DI UNA COLLECTION**

- **CICLO FOR CON INDICE. FUNZIONA SOLO PER LE COLLECTIONS CON ORDINAMENTO. NON SUL SET. PERMETTE LA MODIFICA DURANTE LA VISITA**

```
ArrayList<Integer> array = new ArrayList<>();
for(int i=0; i<DIM; ++i){
    System.out.println( array.get( i ) );
}
```

- **METODO VELOCE**

```
List<Number> listaDiNumeri=new LinkedList();
for (Number n : listaDiNumeri){
    System.out.println(n); //Non è possibile modificare la collection durante la visita
}
```

- **ITERATOR**

```
void filter(Collection<Number> x) {
    Iterator<Number> i = x.iterator();
    while(i.hasNext()) {
        if(!cond(i.next()))
            i.remove(); // rimuove gli elementi della collection che soddisfano una determinata condizione
    }
}
```

## **EQUALS E HASHCODE**

- The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object.
- **LA EQUALS DI OBJECT SI PUÒ SEMPRE CHIAMARE CON == ANCHE SE È STATA RIDEFINITA LA EQUALS**
- Equals dovrebbe essere equivalenza matematica. Riflessiva Simmetrica e Transitiva
- **NOTA SULLE COLLECTIONS. SET RICHIEDE EQUALS E HASHCODE. I LAYOUT IN JAVA FX FUNZIONANO COME I SET, DUNQUE CHIAMANO EQUALS PER CONTROLLARE SE IL NODO È GIÀ PRESENTE**
- **NECESSARIO** @Overriding di equals(Object a);

```
public int hashCode(){
    return objectInside.hashCode();
}
```

```
public int hashCode() {
    int hash = 0; //WORST HASHCODE EVER BUT CORRECT
    return hash;
}
```

- *hashCode* <- riduce il costo del confronto tra due oggetti. Generalmente mappa oggetto->intero. Confronto su interi meno costoso. Nel caso di
- **NECESSARIO** @Overriding di hashCode(); se c'è overriding di Equals
- c1.equals(c2)=>c1.hashCode()==c2.hashCode().
- c1.hashCode()==c2.hashCode()=>!c1.equals(c2)
- When two objects have the same hashCode is necessary equals to test for equality. However if two object have different hashCode they can't be equal
- **Proprietà** di hashCode()
  1. Se invocato più di una volta sullo stesso oggetto deve ritornare lo stesso intero. Questo può essere diverso in esecuzioni diverse dell'applicazione; l'importante è che rimanga identico all'interno di una singola esecuzione.

2. Se due oggetti sono uguali secondo il metodo equals() allora hashCode() deve ritornare lo stesso intero
  3. non è richiesto che a due oggetti diversi (secondo equals()) siano associati due hashCode() diversi
- Minimize collisions

```
public int hashCode(){
    return Object.hash(parameter1,parameter2);
}
```

## ■ ORDINAMENTO COMPARETO

```
class NamedPointComparatorByName implements Comparator {
    public int compare(Object p1, Object p2) {
        NamedPoint np1 = (NamedPoint) p1;
        NamedPoint np2 = (NamedPoint) p2;
        return (np1.getName().compareTo(np2.getName()));
    }
}
//Chiamata
Collections.sort(lista, new NamedPointComparatorByName());
```

- Interfaccia Comparable -> int compareTo(Object b); . La classe deve implementare Comparable e @Override di int compareTo(Object b);
- Negativo se this<b
- Positivo se this>b
- 0 se this = b
- Proprietà
  1. -sgn(x.compareTo(y)) == sgn(y.compareTo(x))
  2. (x.compareTo(y)>0 && y.compareTo(z)>0) => x.compareTo(z)>0
  3. x.compareTo(y)==0 => sgn(x.compareTo(z)) == sgn(y.compareTo(z))
  4. "Consigliato" : (x.compareTo(y)==0) == (x.equals(y))
- **ORDINAMENTO DI STRINGE ALFABETICO**

```
ArrayList <String> testo = new ArrayList<>();
testo.sort(String::compareToIgnoreCase);
```

## ■ COMPARATOR

- Confronto da una altra classe. Permette di definire diversi ordinamenti allo stesso tempo con più classi di confronto.

## ■ WRAPPERS AUTOBOXING

- Boolean<-boolean
- Integer<-int
- Float<-float
- Double<-double
- Char<-char
- Oggetti che incapsulano i tipi semplici <- Collections SOLO su OGGETTI

## ■ ENUM

- Implementano Comparable. Sono "simili" a classi

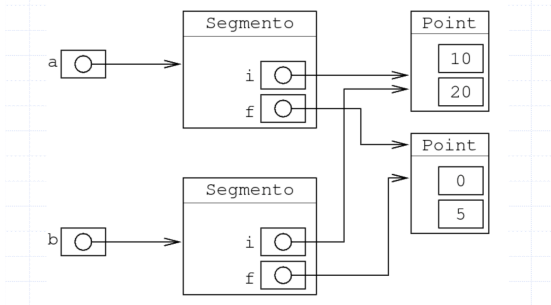
```
Enum Seme{ Cuori, Quadri, Fiori, Picche };
Seme c = Seme.Cuori;
c.name(); //Cuori          c.ordinal() // 0
for( Seme x : Seme.values() ){
    System.out.println(x); //Cuori Quadri Fiori Picche
}
```

```
Classe P implements Cloneable{
    @Override
    public Object clone(){
        Object tmp;
        try{
            tmp= super.clone();
        }catch(CloneNotSupportedException ex){}
        /* add your deep copy code here*/
    }
}
```

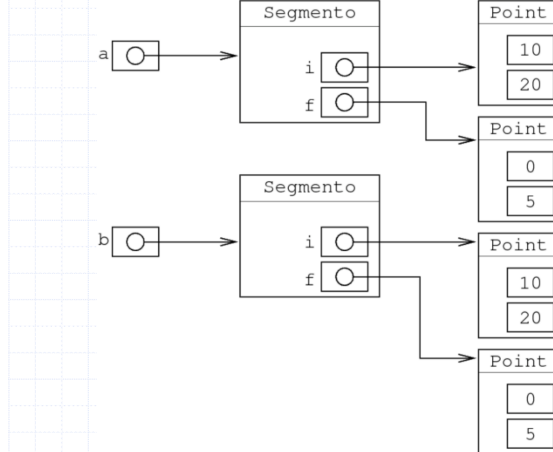
## CLONEABLE

- Object ha una clone. Protected. Default Shallow Copy
- Nel caso sia necessaria una Deep Copy, aggiungere il codice Deep dopo aver effettuato una Shallow Copy
- In JAVAFX un NODE può avere solo un Parent. Per aggiungere lo stesso Node a parent diversi è necessaria la copia

### Copia superficiale (shallow copy)



### Copia profonda (deep copy)



## GENERICICS

- Usate per costruire funzioni/classi parametriche per più tipi( vedi polimorfismo ).

```
class Pair<X,Y> {
    private X first;
    private Y second;
    public Pair(X a1, Y a2) {
        first = a1;
        second = a2;
    }
}
```

- NON** vale Liskov. Dati due tipi generici **G<A>** e **G<B>** dove **B** è una sottoclasse di **A**, **non** è vero che **G<B>** è una sottoclasse di **G<A>**
- NON** è possibile fare Array di generics
- Permette di inferire i tipi

```
Set<E> result = new HashSet<>(s1); // <- diamond operator <>
```

- Wildcard. Qualsiasi cosa

```
Group<?> q = new Group<Student>(); NON -> Group<?> q = new Group<?>();
```

- Generics di sottotipo. Quasi qualsiasi cosa

```
Group<? extends Persona> g = new Group<Student>(); //Persona è OK
```

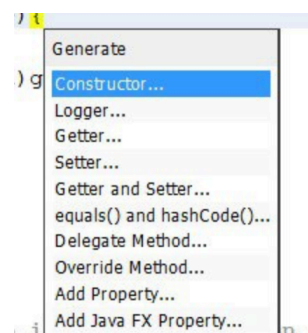
- Generics di supertipo. Quasi qualsiasi cosa


```
Group<? super Studente> g = new Group<Persona>(); //Student è OK
```

## NETBEANS TRICKS

|             |              |          |             |                |              |
|-------------|--------------|----------|-------------|----------------|--------------|
| Fix Imports | Ctrl+Shift+I | Refactor | >           | Insert Code... | Alt+Insert   |
| Refactor    | >            | Format   | Alt+Shift+F | Fix Imports    | Ctrl+Shift+I |
| Format      | Alt+Shift+F  |          |             |                |              |

|            |            |               |
|------------|------------|---------------|
| st Packag  | Run File   | Shift+F6      |
| raries     | Debug File | Ctrl+Shift+F5 |
| st Librari |            |               |



- Usa più main di test. Tasto  e successivamente “run file” sulla classe contenente il main scelto



# JAVAFX

## ■ NODE

- Metodi IMPORTANTI di un node. Un Node può avere al massimo un Parent.

```
node.getChildren() <- ritorna la lista dei figli
node.getParent() <- ritorna il parent in cui è contenuto
node.getChildren().clear() <- rimuove tutto ciò che è contenuto all'interno del node
```

- Inserimento all'interno di un PARENT. **NOTA BENE**

```
node.getChildren().add /addAll( node) <- aggiunge i nodi. Chiama la EQUALS prima di aggiungere. Se si cerca di aggiungere un nodo che secondo la equals è già presente, viene lanciata una eccezione
```

- Il node può “richiedere” al Parent il proprio posizionamento. Sta al Parent decidere se assecondarlo o meno

```
node.setAlignment(Pos.CENTER);
```

- Alcune figure base:

- Circle

```
figura = new Circle(Commons.CELLSIZE/2);
```

- Rectangle

```
figura = new Rectangle(Commons.WIDTH, Commons.HEIGHT);
```

- Polygon

- Triangolo

```
figura = new Polygon();((Polygon)figura).getPoints().addAll(new Double[]{
    Commons.CELLSIZE/2, 0.0 ,
    0.0, Commons.CELLSIZE,
    Commons.CELLSIZE, Commons.CELLSIZE});
```

- Any Perfect Inscribed Polygon

```
Polygon p=new Polygon();
p.getPoints().addAll(gen_points(10)); //<-esempio di chiamata
public Double[] gen_points(int n){ //<- ringraziamenti a Giacomo
    Double[] punti =new Double [2*n];
    double x_in = DIM/2;
    double y_in = 0;
    double r=DIM/2.0;
    punti[0]= x_in;
    punti[1]= y_in;
    double angle = 2*Math.PI/n; //step
    double curr_angle=0;
    for(int i=2;i<2*n;i+=2){
        curr_angle+=angle;
        double x= x_in + (r*Math.sin(curr_angle));
        double y= y_in + (r*Math.cos(curr_angle));
        punti[i]=x;
        punti[i+1]=y;
    }
    return punti;
}
```

- Ruotare una figura. Il posizionamento viene definito prima della rotazione.

```
// Rotazione rispetto al centro
Rotate rot = new Rotate(45, x, y);
figura.getTransforms().add(rot);
```

```
//metodo migliore
nodo.setRotate( angoloInGradi );
```



## COLOR

```
import javafx.scene.paint.Color;
```

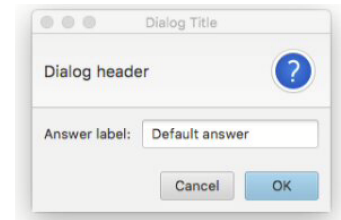
- Set fill && set stroke

```
figura.setFill(Color.YELLOW);          figura.setStroke(Color.RED);
```

## INPUT

- TEXT INPUT DIALOG

```
TextInputDialog dialog = new TextInputDialog("Default answer");
dialog.setTitle("Dialog Title");
dialog.setHeaderText("Dialog header");
dialog.setContentText("Answer label:");
String s= dialog.showAndWait().get();
```



- Ottenimento di input molto carino da parte di Ronchetti

```
import java.util.Optional; import javafx.scene.control.TextInputDialog;
public class InputDialog{
    static String getResponse(){
        TextInputDialog dialog = new TextInputDialog();
        Optional<String> result = dialog.showAndWait();
        if(result.isPresent() ) {    return result.get(); }
        return null;
    }
}
```



- TEXTFIELD

```
textfield.getText(); <- ottiene il testo          textfield.clear(); <- pulisce il testo
```



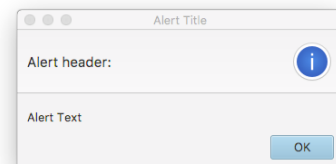
```
textfield.setEditable(false); <- imposta non modificabile dall'utente
```

- TEXT( NOT INPUT THOUGH)

```
Text t = new Text();
text.setFont(new Font(20));
text.setWrappingWidth(200);
text.setText("The quick brown fox jumps over the lazy dog");
```

## ALERT

```
Alert alert = new Alert(AlertType.INFORMATION);
alert.setTitle("Alert Title");
alert.setHeaderText("Alert header:");
alert.setContentText("Alert Text");
alert.showAndWait();
```



- Settaggio di azioni bottone OK

```
Alert alert = new Alert(Alert.AlertType.ERROR, "Ciao", ButtonType.OK);
Optional<ButtonType> result = alert.showAndWait();
if (result.isPresent() && result.get() == ButtonType.OK) {
    // do something
}
```

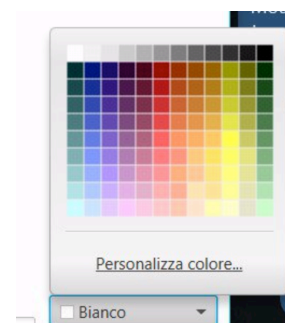
## COLORPICKER

- Metodo veloce con Lambda

```
import javafx.scene.control.ColorPicker;
ColorPicker cp=new ColorPicker();
cp.setOnAction(e -> {nodoDaColorare.setFill(cp.getValue());});
```

- Alternativa con Classe anonima

```
colorPicker = new ColorPicker();
colorPicker.setOnAction( new EventHandler(){
    @Override
    public void handle(Event event) {
        nodoDaColorare.setFill( colorPicker.getValue() );
    }
});
```





## ■ RIPRISTINARE LO START

- basta chiamare start all'interno di start stesso, per esempio dallo handler di un bottone. Tutto ciò che è inizializzato nello start viene inizializzato nuovamente.
- Ciò che è inizializzato nel costruttore dell'applicazione NON viene inizializzato nuovamente

```
this.start(primaryStage); this.close();  
//o in alternativa se necessario aprire una nuova finestra  
this.start(new Stage());
```

## ■ STAGE

- Creazione di Finestre multiple <- multipli stage.

```
Stage newWindow = new Stage(); newWindow.setTitle(Commons.TITLE);  
//nel caso aggiungere un Layout se necessario  
Scena = new Scene( node, Commons.CONTROLLERWIDTH, Commons.CONTROLLERHEIGHT);  
newWindow.setScene(scena);  
newWindow.show();
```

- Disabilitare ridimensionamento

```
primaryStage.setResizable(false);
```

## ■ Immagini, AUDIO, MEDIA e Files

```
public class Sounds extends Application { public void start(Stage stage) {  
    Media media = new Media("http://www.ferraterreaacqua.it/ it/audioguide/audioguide-di-ferrara-citta-del-rinasci-  
mento/ 01_benvenuto-a-ferrara.mp3");  
    MediaPlayer mediaPlayer = new MediaPlayer(media);  
    mediaPlayer.setAutoPlay(true); // create mediaView and add media player to the viewer MediaView  
    mediaView = new MediaView(mediaPlayer);  
    Group root = new Group(mediaView);  
    root.getChildren().add( new Text(10, 30, "Benvenuto a Ferrara"));  
    Scene scene = new Scene(root, 150, 60);  
    stage.setScene(scene);  
    stage.sizeToScene();  
    stage.show();  
} public static void main(String[] args) { Application.launch(args); }  
}
```

## ■ IMAGE

```
public class FilesAndImages extends Application {  
    @Override  
    public void start(Stage stage) {  
        FileChooser fileChooser = new FileChooser();  
        fileChooser.setTitle("Carica un'immagine");  
        fileChooser.getExtensionFilters().addAll( new FileChooser.ExtensionFilter("JPG", "*.jpg"), new  
            FileChooser.ExtensionFilter("PNG", "*.png") );  
        String url = System.getProperty("user.home");  
        File f=new File(url);  
        fileChooser.setInitialDirectory(f); // bugged on MacOSX  
        File file = fileChooser.showOpenDialog(stage);  
        if (file == null) { System.out.println("No file chosen");  
            System.exit(1);  
        }  
        // Image & File  
        Image image = new Image("file://" + file.getAbsolutePath(), 500, 500, true, true);  
        ImageView iw = new ImageView(image);  
        Group root = new Group(iw);  
        Scene scene = new Scene(root, 500,500);  
        stage.setTitle(file.getName());  
        stage.setScene(scene);  
        stage.sizeToScene();  
        stage.show();  
    }  
    public static void main(String[] args) { Application.launch(args); }  
}
```

## ■ LAYOUT PREDEFINITI

- SetLayout di un Nodo all'interno del Parent è deciso dal Parent stesso. SetTranslate è deciso dal Nodo
- La posizione è data da : SetLayout + SetTranslate.
- The node's final translation will be computed as **layoutX + translateX**, where layoutX establishes the node's stable position and translateX optionally makes dynamic adjustments to that position.
- NON FUNZIONA QUASI MAI. Sta al parent decidere se Assecondare le richieste del figlio

*node.setAlignment( Pos.BOTTOM\_CENTER); <- non usare! Il posizionamento effettivo è deciso dal parent e non dal node stesso*

#### ■ **HBOX** The HBox class arranges its content nodes horizontally in a single row.

- Dividere equamente lo spazio tra i figli

```
HBox hbox = new HBox(); <-//nel costruttore è possibile passare lo spacing intero e i figli
Button button1 = new Button("Add");
Button button2 = new Button("Remove");
HBox.setHgrow(button1, Priority.ALWAYS);
HBox.setHgrow(button2, Priority.ALWAYS);
button1.setMaxWidth(Double.MAX_VALUE);
button2.setMaxWidth(Double.MAX_VALUE);
hbox.getChildren().addAll(button1, button2);
```

- Spacing <- spazio bianco tra i figli

*hbox.setSpacing(double value)*

- CENTRATURA

*hbox.setAlignment(Pos.CENTER); //richiede al parent di essere centrato E di conseguenza centra i figli*

#### ■ **VBOX** The VBox class arranges its content nodes vertically in a single column.

- Dividere equamente lo spazio tra i figli

```
VBox vbox = new VBox(); <-//nel costruttore è possibile passare lo spacing intero e i figli
Button button1 = new Button("Add");
Button button2 = new Button("Remove");
VBox.setVgrow(button1, Priority.ALWAYS);
VBox.setVgrow(button2, Priority.ALWAYS);
button1.setMaxHeight(Double.MAX_VALUE);
button2.setMaxHeight(Double.MAX_VALUE);
vbox.getChildren().addAll(button1, button2);
```

- Spacing tra i figli

*vbox.setSpacing(double value)*

- CENTRATURA

*vbox.setAlignment(Pos.CENTER); //richiede al parent di essere centrato E di conseguenza centra i figli*

#### ■ **STACKPANE** The StackPane class places its content nodes in a back-to-front single stack.

- StackPane centra automaticamente i nodi.
- Set Alignment di un nodo all'interno di dello StackPane

*StackPane.setAlignment( button, Pos.BOTTOM\_CENTER);*

- In generale lo stackPane ha dimensioni "liquide". Usare setMin/Max in caso di necessità di avere dimensioni FISSE

```
stackPane.setMinWidth( dimensione);    stackPane.setMaxWidth( dimensione);
stackPane.setMinHeight( dimensione);    stackPane.setMaxHeight( dimensione);
```

#### ■ **TILEPANE** The TilePane class places its content nodes in uniformly sized layout cells or tiles

- Set del numero di colonne / righe

*tile.setPrefColumns(4);<- set del numero di colonne tile.setPrefRows(4);<- set del numero di righe*

- Set della distanza tra i figli

*tile.setVgap(8);<-distanza verticale tra i figli tile.setHgap(4); // <- distanza orizzontale tra i figli*

#### ■ **FLOWPANE** The FlowPane class arranges its content nodes in either a horizontal or vertical "flow," wrapping at the specified width (for horizontal) or height (for vertical) boundaries.

- Settare spaziatura prima di andare a capo

*flow.setPrefWrapLength(300); // preferred width = 300*

- Set distanza tra i figli

```
flow.setVgap(8); // <- distanza verticale tra i figli    flow.setHgap(4); // <- distanza orizzontale tra i figli
```

- FlowPane verticale da aggiungere

■ **BORDERPANE** The BorderPane class lays out its content nodes in the top, bottom, right, left, or center region.

- Inserimento di elementi di un BorderPane

```
controlButtons.setTop(nodo);
controlButtons.setRight(nodo);
controlButtons.setBottom(nodo);
controlButtons.setLeft(nodo);
controlButtons.setCenter(nodo);
```

- Set alignment di un nodo all'interno di un borderpane. Il borderPane asseconda "abbastanza spesso" le esigenze del figlio

```
BorderPane.setAlignment(nodo, Pos.CENTER); oppure node.setAlignment(Pos.CENTER);
```

■ **ANCHORPANE** The AnchorPane class enables developers to create anchor nodes to the top, bottom, left side, or center of the layout.

Da aggiungere

■ **GRIDPANE** . The GridPane class enables the developer to create a flexible grid of rows and columns in which to lay out content nodes.

- SELEZIONE ELEMENTO DA GRIDPANE

```
private Node getItemAt (int column, int row){
    for (Node e : this.getChildren()) {
        try{
            if( GridPane.getRowIndex(e) == row && GridPane.getColumnIndex(e) == column){
                return e;
            }
        }catch(NullPointerException ex){
        }
    }
    return null;
}
```

- SWAP DI ELEMENTI DAL GRIDPANE

```
private void swap2Cells(Node a, Node b){
    int xa = GridPane.getColumnIndex(a);
    int ya = GridPane.getRowIndex(a);
    int xb = GridPane.getColumnIndex(b);
    int yb = GridPane.getRowIndex(b);
    /*GridPane.setColumnIndex(a,xb);
    GridPane.setRowIndex(a,yb);
    GridPane.setColumnIndex(b,xa);
    GridPane.setRowIndex(b,yb);*/ probabilmente superfluo. Da provare se funziona
    this.getChildren().remove(a);
    this.getChildren().remove(b);
    this.add(a, xb, yb);
    this.add(b, xa, ya);
}
```

- MOSTRARE BORDI GRIDPANE

```
gridpane.setGridLinesVisible(true);
```

- Aggiunta di un elemento di una posizione specifica

```
GridPane.add( node, row, column); //probably row and column are inverted
```

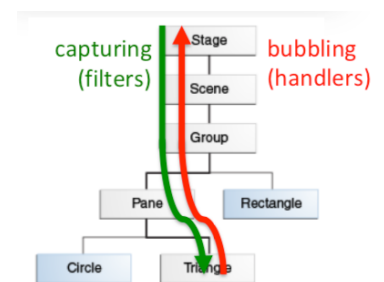
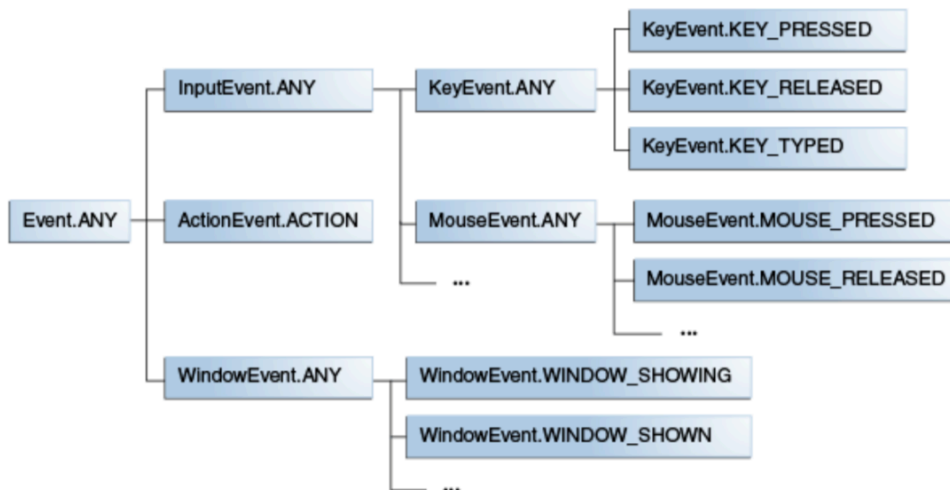
- SETTARE COLONNE E RIGHE FISSE DEL GRIDPANE

```

for (int i = 0; i < Commons.NUMCOLS; i++) {
    ColumnConstraints colConst = new ColumnConstraints();
    colConst.setPercentWidth(100.0 / Commons.NUMCOLS);
    this.getColumnConstraints().add(colConst);
}
for (int i = 0; i < Commons.NUMROWS; i++) {
    RowConstraints rowConst = new RowConstraints();
    rowConst.setPercentHeight(100.0 / Commons.NUMROWS);
    this.getRowConstraints().add(rowConst);
}

```

## EVENTI



- Events can be fired. A Handler implements EventHandler. A Listener implements EventListener
- Consume an event

```
event.consume(); // once consumed, the event will be deleted from the ChainEvent
```

- Listener Vs Handler
- Listener prima, Handler dopo. Target e Source sono diversi, vedi API
- Un NODE può avere più handler/listener anche per lo stesso evento. Nel caso verranno eseguiti in maniera sequenziale a seconda di quale viene per prima nel codice.
- 4 modalità
- Listener integrato <- bad idea Netbeans Warning

```

public class AppWithEvents extends Application implements EventHandler {
    public void start(Stage stage) {
        Button btn = new Button();
        btn.addEventHandler(ActionEvent.ACTION, this);
        Group root = new Group(btn);
        root.getChildren().add(text);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.show();
    }
    public void handle(Event t) {
        //fai qualcosa. Può accedere a tutti i campi di AppWithEvents
    }
}

```

- Listener interno <- OK need reference probably

```

private class Bottone extends Button{
    // cose del bottone
    Dichiarazione della classe handler qui ->
}

```

```

private class HandlerAvvia implements EventHandler{
    @Override
    public void handle(Event event) {
        //fai qualcosa
        //Bottone.this serve ad accedere ai parametri this di Bottone
        Bottone.this.setDisable(true);
    }
}

```

- Listener interno anonimo <- OK can be used without reference. Utile per i bottoni


```
btn.setOnAction(MouseEvent.MOUSE_CLICKED, new EventHandler(){
    public void handle(Event t) {
        //fai qualcosa. Può accedere a tutti i campi sopra del contenitore in cui è contenuta questa riga di co-
        dice
    }
});
```

- Listener esterno <- OK need reference

```
class Listener implements EventHandler {
    private Object ref
    public Listener(Object ref){
        this.ref=ref
    }
    public void handle(Event t) {
        //accedi ai campi di ref se necessa-
        rio fare modifiche
    }
}
```

```
//uso
Listener a = new Listener(this);
btn.addEventHandler(ActionEvent.ACTION, a);
```

## ■ Λ-EXPRESSIONS

- Il compilatore permette di evitare di scrivere tanto codice e inferisce cosa da fare in base alla compatibilità tra tipi. Netbeans permette di passare tra λ-Expressions e Classe interna anonima senza alcuna fatica col tasto 
- Inferring the functional interfaces
  - Does the interface have only one abstract (unimplemented) method?
  - Does the parameters (types) of the lambda expression match the parameters (types) of the single method?
  - Does the return type of the lambda expression match the return type of the single method?
- Esempio con classe interna anonima.

```
c.setOnMouseEntered(new EventHandler<MouseEvent>() {
    public void handle(MouseEvent event) {
        System.out.print("Entered"); c.setFill(Color.RED);
    }
});
```

- Esempio con lambda expression equivalente.

```
c.setOnMouseEntered(( MouseEvent event) -> { System.out.print("Entered"); c.setFill(Color.RED); });
```

## ■ FIREEVENTS

- Permette di simulare un Evento, facendo scattare il handler associato

```
private class KeyHandler implements EventHandler{
    @Override
    public void handle(Event event) {
        Button tmp = new Button("Premimi");
        tmp.setOnAction( new buttonHandler() );
        //nota fireEvent ha bisogno di un event dello stesso tipo dell'eventhandler associato.
        tmp.fireEvent( new ActionEvent() ); <- fireEvent lancia un evento su tmp, che è dello stesso tipo per cui
        implementa già un handler. Dunque viene chiamato buttonHandler();
    }
}
```

## ■ BUTTON

- BOTTONE CHE OCCUPA TUTTA LA LARGHEZZA

```
button2.setMaxWidth(Double.MAX_VALUE); da provare button2.setMinWidth(Double.MAX_VALUE);
```

- DISABILITARE

```
button.setDisable(false);
```

## ■ OBSERVABLE LIST

- Sono una collection che permette di ricevere notifiche se avvengono modifiche alla stessa.

```
ObservableList<Object> lista = FXCollections.observableArrayList();
lista.add(elemento);
lista.addListener(new ListChangeListener(){
    @Override
    public void onChanged(ListChangeListener.Change change) {
        // fai quello che devi fare se c'è stata una modifica
    }
});
```

## ■ ACTION EVENTS

- Sono gli event di default per il button.setOnAction(). NON e' proprio la stessa cosa di MouseEvent. In particolare nel caso di fire, è necessario il fire dell'evento specifico per il handler

```
button.setOnAction(new EventHandler(){
    @Override
    public void handle(Event event){
    }
});
```

- E' quasi equivalente a

```
button.addEventHandler(MouseEvent.MOUSE_CLICKED, new EventHandler(){
    @Override
    public void handle(Event event){
    }
})
```

- La versione col mouseCLICKED consuma l'evento. I bottoni consumano l'evento MouseEvent ma non l'evento ActionEvent. Risolvere utilizzando la skin

```
btn.setSkin(new ButtonSkin(btn) {
    {
        this.consumeMouseEvents(false);
    }
});
```

## ■ KEYEVENTS

- Verificare che un dato tasto sia stato premuto.

```
((KeyEvent)event).getCode()==KeyCode.ENTER;
```

## ■ MOUSEEVENTS

- Ottenere posizione X Y del mouse quando l'evento è stato avviato

```
((MouseEvent)event).getX(): ((MouseEvent)event).getY():
```

- Verificare che il click sia soltanto sinistro

```
public void handle(MouseEvent event) {
    if (event.getButton() != event.getButton().PRIMARY){
        return;
    }
}
```

- **NOTA.** FireEvent su un action MouseEvent è estremamente complicato. Meglio utilizzare un button con un actionEvent. Comunque se sei coraggioso, prego
- Dalle API:
- MouseEvent(Object source, EventTarget target, EventType<? extends MouseEvent> eventType, double x, double y, double screenX, double screenY, MouseButton button, int clickCount, boolean shiftDown, boolean controlDown, boolean altDown, boolean metaDown, boolean primaryButtonDown, boolean middleButtonDown, boolean secondaryButtonDown, boolean synthesized, boolean popupTrigger, boolean stillSincePress, PickResult pickResult)
- In maniera più pragmatica

```
nodoTarget.fireEvent( new MouseEvent(MouseEvent.MOUSE_CLICKED,coordinateX, coordinateY,
posizionamentoAssolutoX, posizionamentoAssolutoY, MouseButton.PRIMARY, 1,true, true, true,
true, true, true, true, true, true, true, true, null) );
```

- Oppure richiamando il metodo statico

```
Event.fireEvent( nodoTarget, new MouseEvent(MouseEvent.MOUSE_CLICKED,  
coordinateX, coordinateY, posizionamentoAssolutoX, posizionamentoAssolutoY, MouseButton.PRIMARY, 1,true, true, true, true, true, true, true, true, true, null));
```