OpenLaszlo:
Una breve introduzione

Cos'è OpenLaszlo?

- E' la **piattaforma di sviluppo** di riferimento per la progettazione di RIA

- E' basato su un **linguaggio di marcatura** per la descrizione dell'interfaccia

- E' composto da un **server** che "legge" queste descrizioni e "costruisce/compila" l'interfaccia "disegnata" dall'utente

Perchè proprio OpenLaszlo?

Open-source

(ma la licenza permette di rivendere le applicazioni create senza problemi)

Basato su Adobe Flash plugin

- *Si stima che Flash Player plugin sia installato su circa il 97% dei pc*
- *Identica resa grafica/funzionale su qualsiasi web-browser*

Server multi-piattaforma

Il server per "compilare" le GUI gira su Windows, Linux e Mac Os X

Prodotto maturo

Prodotto stabile, ottima documentazione, comunità molto attiva

Cosa si può ottenere

- OpenLaszlo demo:
  - World Clock
  - LxPix
  - eCommerce Store
- Altre RIA degne di nota:
  - http://book.orzar.net, interfaccia per Amazon
  - http://www.cooqy.com, interfaccia ad eBay
  - www.gliffy.com, *disegno di diagrammi / flow-chart*
  - http://g.ho.st, *vero e proprio sistema operativo online*

Come funziona in breve

1. Si "*disegna*" l'interfaccia grafica per mezzo di un **linguaggio di marcatura** (*LZX*) basato sullo standard XML
2. Si "*passa*" questa descrizione al **server OpenLaszlo** il quale la "*compila*" e crea l'interfaccia in uno dei seguenti formati:
   - file **SWF** (formato di Adobe Flash);
   - classica pagina **DHTML** (HTML + DOM + JAVASCRIPT).

3. L'oggetto così creato, cioè la nostra interfaccia, viene quindi caricata su un **qualsiasi web-server** *(es. Apache, Tomcat, IIS)*
4. A questo punto è possibile utilizzare la nostra interfaccia grafica attraverso un **qualsiasi web-browser**:

- *raggiungendo quindi il 97% dei pc connessi in rete;*
- *ottenendo una visualizzazione coerente e corretta indipendentemente dal browser utilizzato;*
- *risparmiando il tempo speso abitualmente per ottenere gli stessi effetti/layout su browser diversi e non compatibili.*

Generazione della GUI

DEPLOYMENT OPTIONS

1. LZX APPLICATION
2. OPENLASZLO COMPILER/SERVER
3. "PROXIED" DEPLOYMENT (SWF OR DHTML) / SOLO DEPLOYMENT (SWF OR DHTML)
4. FLASH PLAYER / BROWSER

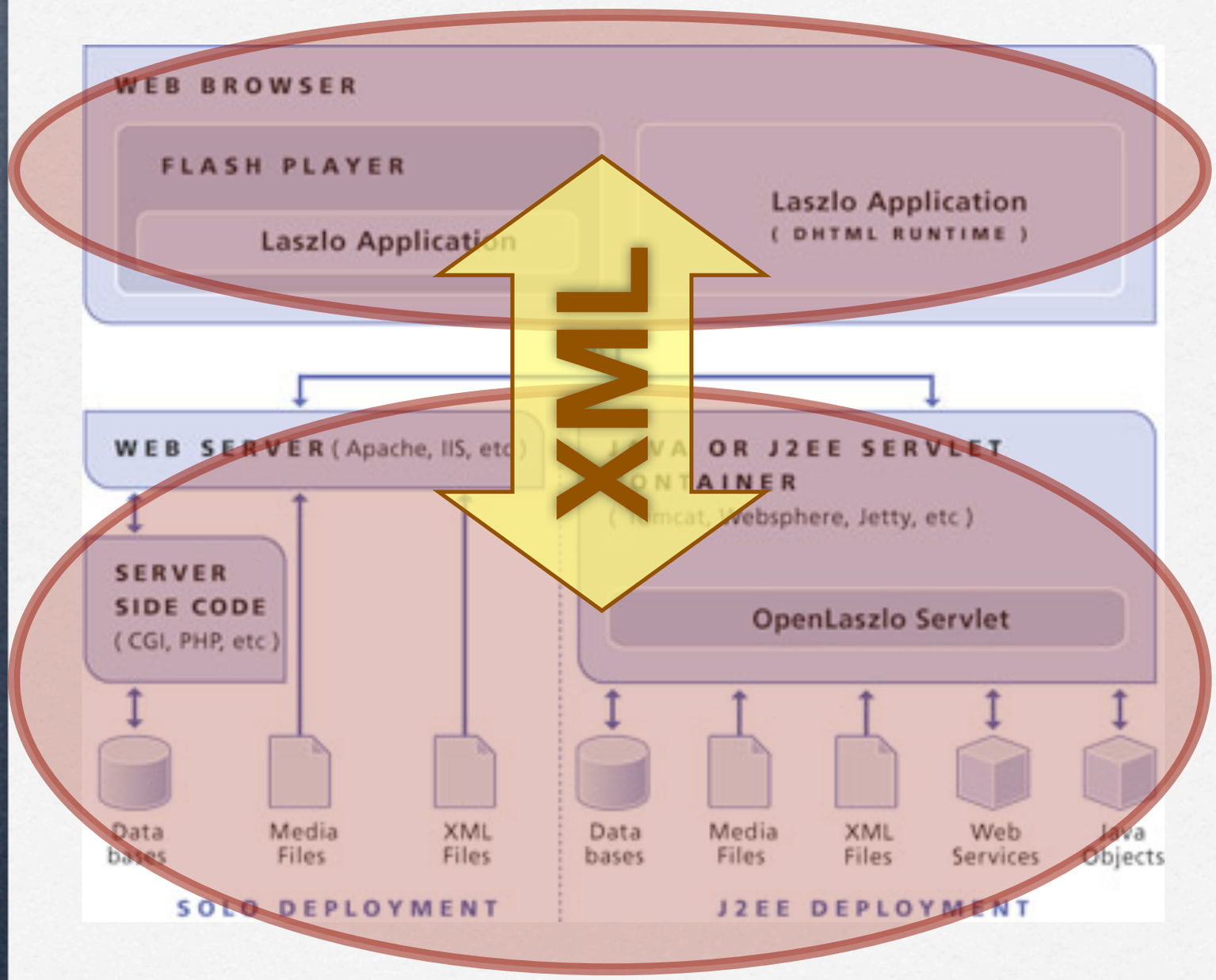
PROXIED
(sviluppo)

SOLO
(distribuzione)

Proxied Deployment:
Bi-directional communication between LZX Client Application and OpenLaszlo Server

SOLO Deployment:
Once the Application is deployed, no further communication to OpenLaszlo Server

Architettura



front-end

=

OpenLaszlo RIA

back-end

=

QUALSIASI !!!!

Il linguaggio LZX

- **Linguaggio di marcatura** proprietario usato per descrivere l'interfaccia in *OpenLaszlo*
- LZX = **L**as**Z**lo **X**ML
- Estensione di **XML**, quindi attenzione a maiuscole/minuscole, spazi, caratteri di escape (es. <) ecc
- Linguaggio **OO**: *classi*, *oggetti*, *metodi, ereditarietà*
- **Programmazione ad eventi**
  (es. onclick, onload, onerror ecc)

- Permette di descrivere sia il **layout** della GUI ma anche la sua **logica** di funzionamento:

- **Layout** = "*com'è la mia interfaccia?*"
  - quali componenti ha la mia interfaccia?
    (finestre, pulsanti, griglie, slider, menù ecc)
  - quali caratteristiche hanno?
    (colore, dimensione, posizione, trasparenza ecc)

- **Logica** = "*cosa succede se...*"
  - ...se l'utente clicca sul pulsante?
  - ...quando l'immagine è stata caricata?
  - ...se nel caricamento si è verificato un errore?

Documentazione

- Sul sito **www.openlaszlo.org** è presente una vasta e dettagliata documentazione:
  - **Developer's Guide**
    Guida ai concetti principali e allo sviluppo con OpenLaszlo
  - **Reference Manual**
    Manuale di riferimento per il linguaggio LZX
- C’è inoltre un forum di discussione in cui si possono trovare consigli molto utili

OpenLaszlo:
Come definire il LAYOUT

Tag e attributi

- Proprio come *HTML*, anche *LZX* è basato su *tag* e *attributi*
- I **tag** definiscono i componenti/widget da posizionare sulla GUI (finestre, pulsanti, immagini...)
  - *Ad es. l'intera applicazione è contenuta nel tag:*
    <canvas> ..... **</canvas>**
- Gli **attributi** (*proprietà*) definiscono invece l'aspetto visuale (colore, posizione, dimensione...)

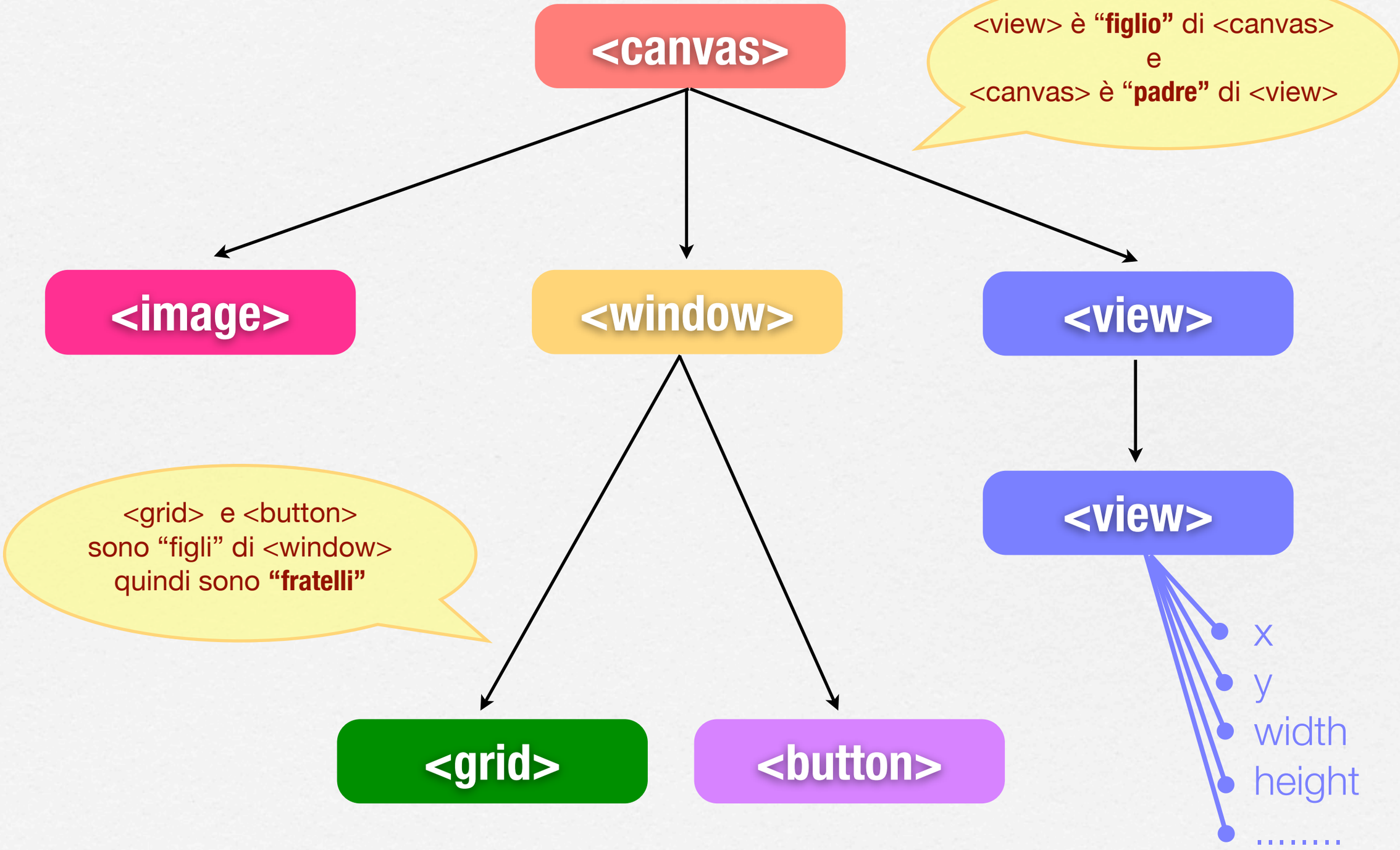
- I **tag** possono essere scritti in 2 forme:
  - **<TAG** …attributi… **>** …altri tag annidati… **</TAG>**
  - **<TAG** …attributi… **/>**
- Anche per gli **attributi** esistono 2 sintassi:
  - <TAG nomeAttributo**=**”valoreAttributo” >
  - <TAG>
    <attribute name=”nomeAttributo”
    type=”tipoAttributo”
    value=”valoreAttributo”
    />
    </TAG>

Document Object Model

- Per accedere agli elementi dell'interfaccia si usa il **DOM** (**D**ocument **O**bject **M**odel)

- E' una **rappresentazione ad albero** della struttura dell'interfaccia (*tag* e *attributi*)

- E' utilizzata per accedere e aggiornare dinamicamente il contenuto, la struttura e lo stile dei componenti dell'interfaccia


<canvas>
<view> è “figlio” di <canvas>
e
<canvas> è “padre” di <view>
<image>
<window>
<view>
<view>
<grid> e <button>
sono “figli” di <window>
quindi sono “fratelli”
x
y
width
height
.....
<grid>
<button>

- Ogni oggetto può essere identificato per mezzo di un **nome locale/globale**:
  - attributo *id*: scope globale a tutta l'applicazione
  - attributo *name*: *scope locale al "padre" che lo contiene*

- Si usa la classica **sintassi puntata** "." per accedere a oggetti e proprietà
  (es. *myWindow.width*, *myButton.xoffset* ecc)

- Si può navigare nella gerarchia anche con **parole chiave**: *canvas*, *this*, *parent*, *subviews*…
  (es. *canvas.myWindow.x*, *this.bgcolor*, *myButton.parent* ecc)

Il tag <view>

- E' l'oggetto di base di *OpenLaszlo*
- Consiste in un **box rettangolare**
  (equivalente a MovieClip in Flash e in HTML)
- **Contenitore** per oggetti multimediali
  (immagini, audio, video ecc)
- Elemento visuale di base grazie al quale costruire componenti complessi
- **Tag annidati**: sistema di riferimento è relativo al "padre" (angolo alto-sinistra)

Principali attributi

- Posizione e dimensione:
  width, *height*, *x*, *y*, *rotation*, *xoffset*, *yoffset*
- Aspetto:
  bgcolor, *fgcolor*, *opacity*, *visible*, *clip*
- Risorse multimediali:
  source, *resource*, *stretches*
- *Risorse video/animazioni Flash:*
  frame, *totalframes*, *playing*

Risorse multimediali

- E’ possibile caricare **risorse multimediali** (immagini, suoni, video) dentro una *<view>* per mezzo degli attributi *source*/*resource*
- Una risorsa può essere caricata:
  - a tempo di esecuzione (**run-time**)
    La risorsa viene caricata solamente al momento dell’inizializzazione della view che la contiene
  - a tempo di compilazione (**compile-time**)
    La risorsa viene inglobata nel binary dell’applicazione ed è quindi inclusa nel download iniziale dell’applicazione stessa

- Caricamento a **run-time**:
  - Si specifica la risorsa da caricare come **url HTTP**:
    <view **resource=”http:**//www.sito.com/immagine.jpeg**”** />
    <view **resource=”http:**../icone/disk.jpeg**”** />
  - Oppure usando l’attributo **source** *(è sempre run-time)*:
    <view **source=”**www.sito.com/immagine.jpeg**”** />
  - La risorsa viene caricata solo al momento dell’inizializzazione della *view* oppure quando l’attributo *resource* cambia valore
  - Il download iniziale dell’applicazione è più veloce, ma risorsa potrebbe tuttavia essere visualizzata con un ritardo a causa di rallentamenti nella rete

- Caricamento a **compile-time**:
  - La risorsa viene caricata attraverso un tag *<resource>* ausiliario *(identificato da un nome)* oppure specificando nell'attributo *resource* il path locale del fileda caricare:
  - **<resource name="**nomeRisorsa**" src="**media/icon.jpeg**"/>**
    <view **resource="**nomeRisorsa**"** />
    ..... oppure .....
    <view **resource="**../icone/disk.jpeg**"** />
  - La risorsa viene inglobata nell'applicazione stessa, quindi i tempi di download sono maggiori ma la visualizzazione della risorsa sarà istantanea
  - Ideale per risorse usate spesso (icone, suoni ecc)

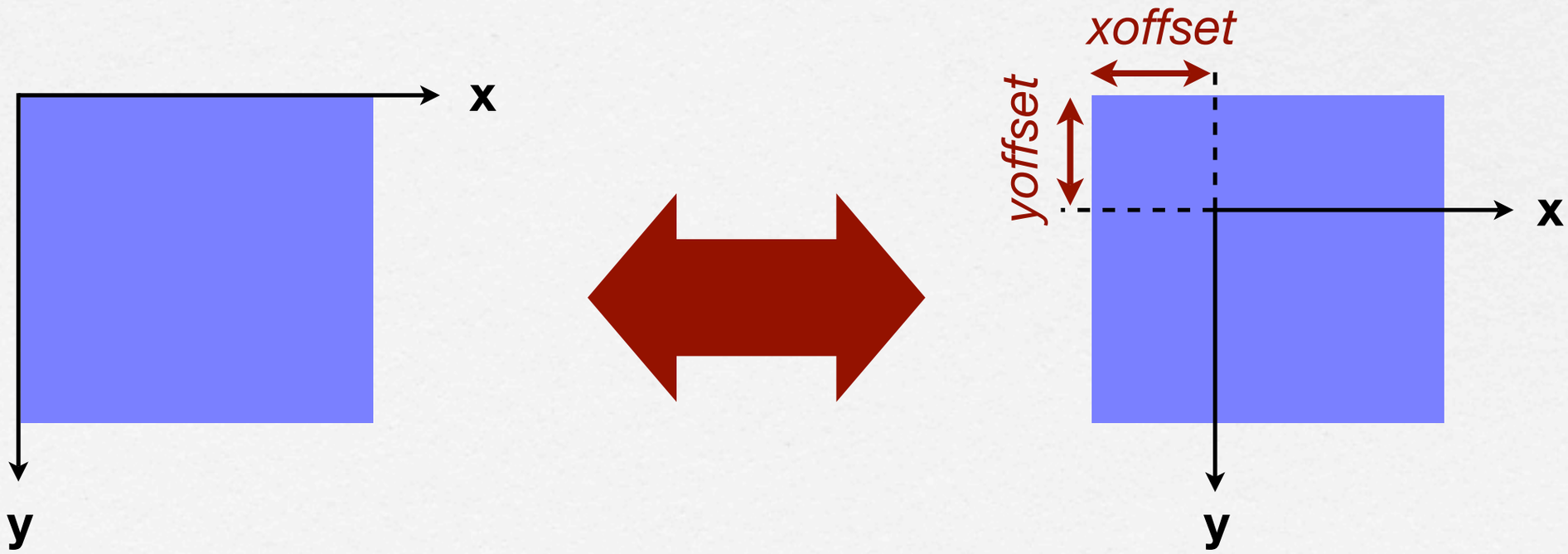
- Con il tag *<resource>* è possibile caricare **risorse multi-frame**, ovvero composte da più sotto-risorse:
  - Le sotto-risorse sono memorizzate in **frame** separati attraverso uno o più tag *<frame>*
  - Si accede ai vari frame per mezzo dell'attributo *frame*:

```
<resource name="mylcons">
    <frame src="resources/pippo.jpeg" />
    <frame src="resources/pluto.jpeg" />
</resource>

<view resource="mylcons" frame="1" />
<view resource="mylcons" frame="2" />
```

Note su *xoffset*/*yoffset*

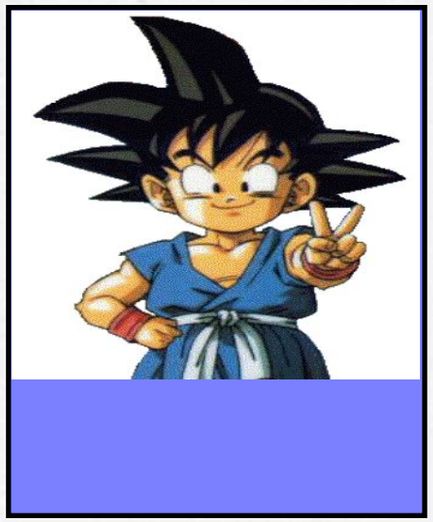
Normalmente il **sistema di riferimento** di un oggetto è relativo all'*angolo in alto a sinistra*. *xoffset* e *yoffset* permettono di spostarlo:



Note su *stretches*

Quando si carica una immagine si possono forzare le sue dimensioni ad **adattarsi al suo contenitore**:

“none”



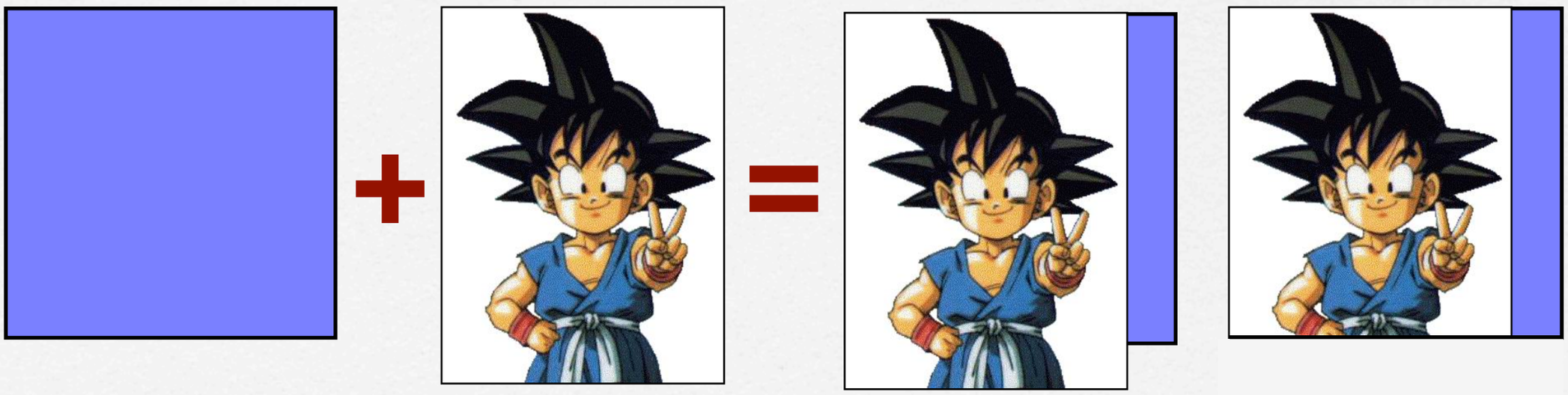
“width”

“height”

“both”

Note su *clip*

Quando il contenuto di un oggetto eccede le dimensioni dell'oggetto stesso si può **nascondere il contenuto in eccesso**:



"false"

"true"

I tag <layout>

- Sono **elementi funzionali** che servono per **distribuire ed allineare automaticamente** i tag “figli” di un oggetto (es. <view>)

- Esempi:

  <simplelayout axis=”x” spacing=”10” **/>**

  Allinea tutti i componenti lungo l’asse *x*, con una spaziatura costante di 10 pixel tra di essi

  <wrappinglayout axis=”x” spacing=”10” **/>**

  Come prima, ma arrivato al margine destro dell’oggetto padre, manda a capo i restanti componenti sulla riga sottostante in automatico

Le classi

- E’ possibile definire **tag personalizzati**:
  <class name=”nomeNuovoTag**”>**
  …definizione degli attributi e dei metodi…
  </class>
- Poi si utilizzano come di consueto:
  <nomeNuovoTag …attributi… **>**
  …altri tag/compnenti annidati…
  </nomeNuovoTag>

- Esempio:

```
<view bgcolor=”red” width=”50” height=”50” x=”0” />
<view bgcolor=”red” width=”50” height=”50” x=”100” />
<view bgcolor=”red” width=”50” height=”50” x=”200” />
```

...si può riscrivere come...

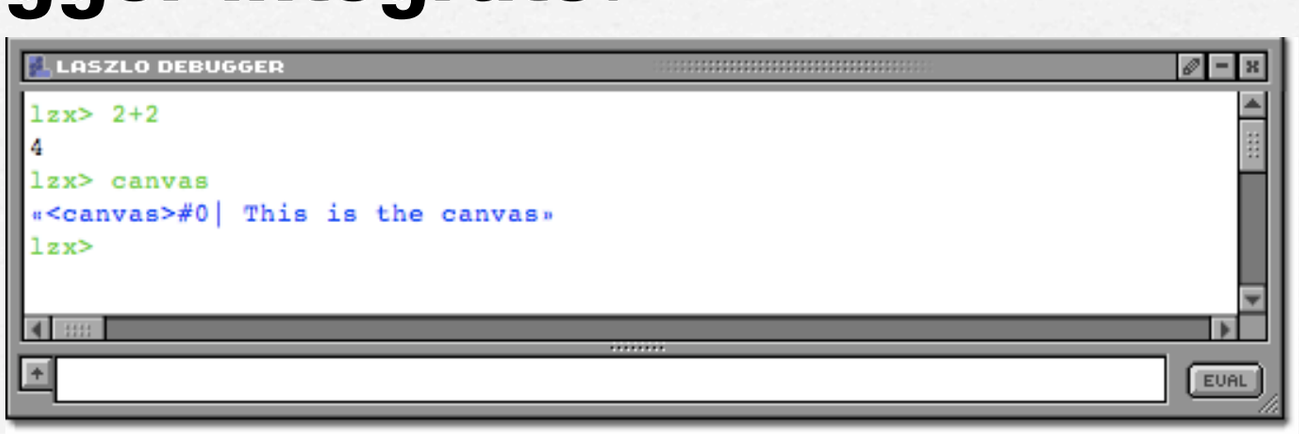
```
<class name=”myView”
      bgcolor=”red” width=”50” height=”50”
/>

<myView x=”0” />
<myView x=”100” />
<myView x=”200” />
```

- Nella definizione è possibile inserire:
  - **attributi**: proprietà/caratteristiche dell'oggetto ← **Layout**
    <attribute name="....." type="....." value="....." />
  - **metodi**: funzioni che l'oggetto può compiere ← **Logica**
    <method name="....." > ..... </method>
  - **eventi**: porzioni di codice da eseguire al verificarsi di particolari condizioni (es. al click del mouse, al caricamento di un'immagine...)
    <handler name="....." > ..... </method>
- Le classi possono essere definite in **file esterni** attraverso il tag *<library>....</library>*. E' poi possibile includerle nel *<canvas>* tramite il tag *<include href="*...path al file lzx...*" />*

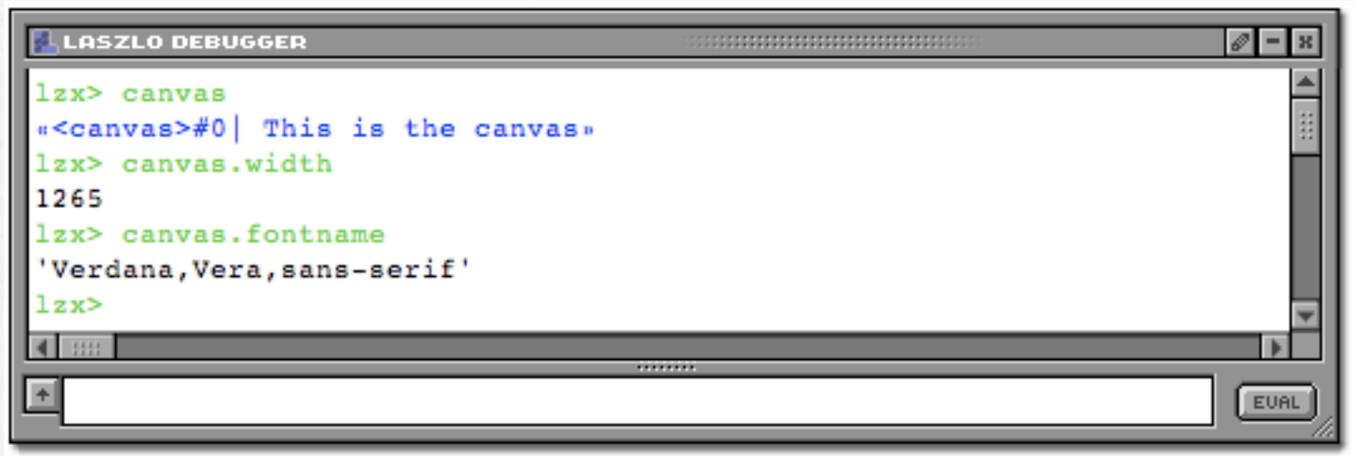
Debugger

- OpenLaszlo mette a disposizione un **debugger integrato**:



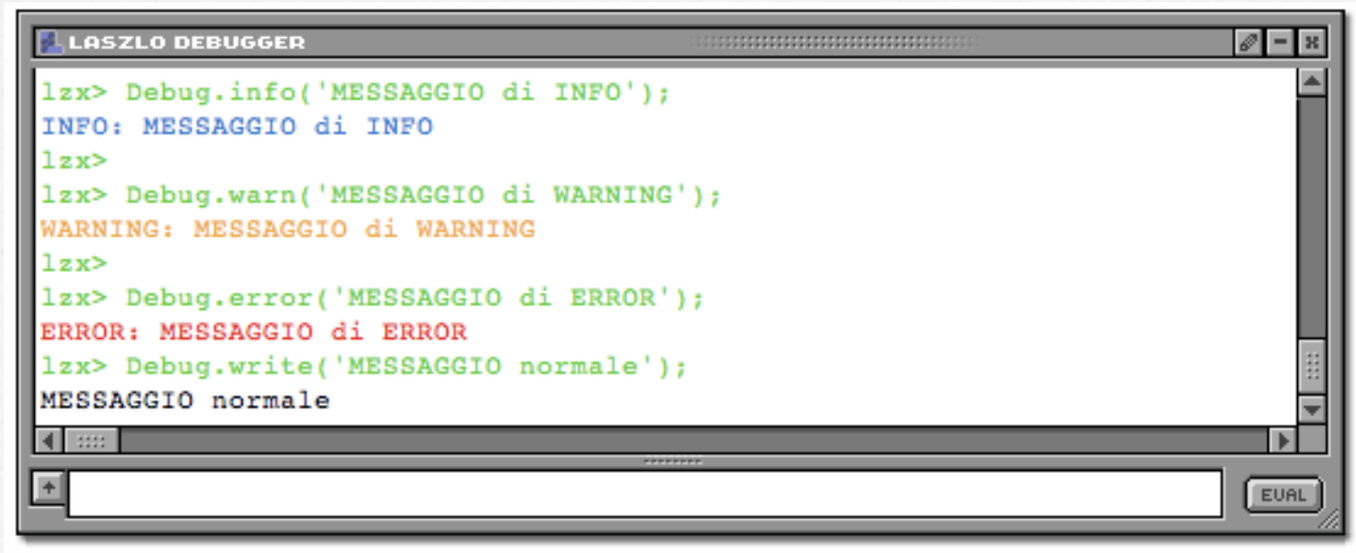
- Per abilitarlo è sufficiente settare l'attributo debug="true" del canvas
- Disponibile solo se si è connessi al server *(cioè in modalità proxied)*

- **Ispezionare** oggetti, attributi, espressioni:



es:
canvas
canvas.width / 2
this.bgcolor
Math.max(this.x,parent.x)

- Stampare **messaggi** informativi:



es:
Debug.info(‘....’)
Debug.warn(‘...’)
Debug.error(‘...’)
Debug.write(‘....’)

Per fare delle prove

1) Andate sul sito www.openlaszlo.org
2) DOCUMENTATON -> Reference
3) Cercare l'help del tag <view>
4) Nel primo esempio, cliccate "EDIT"
5) A questo punto avrete a disposizione un **editor/compilatore online** per fare i vostri esperimenti

1° esercitazione

1. Inserire delle **<view>** sulla GUI, modificare gli attributi di posizione, dimensione e aspetto, e vedere gli effetti che si ottengono
2. Sovrapporre due o più <view> e vedere l'effetto della proprietà **opacity**
3. Provare a replicare degli oggetti con caratteristiche simili con l'uso delle **classi**
4. Posizionare alcuni componenti a scelta con/senza l'aiuto del tag **simplelayout**

I constraints

- Servono a **vincolare il valore di un attributo** a quello di altri attributi *(anche di altri oggetti)*
- Sintassi:

  <tag …… attributo**=”${** espressione **}”** …… **/>**
- Esempio:

```
<view width=”50” height=”${this.width}”>
        <view width=”${parent.width/2}” height=”${parent.height/3}” />
</view>
```

- Molto comodi e pratici, ma non abusarne

  (un uso esagerato degrada le prestazioni dell’interfaccia)

Il tag <animator>

- Serve per **“animare” un oggetto** in un certo intervallo di tempo, ad es.
  - muovere l’oggetto:
    incrementare/diminuire i suoi attributi *x* o *y*
  - ottenere effetti di fading:
    abbassare gradualmente l’attributo *opacity*
  - ma è possibile “animare” **qualsiasi attributo** numerico!
- Sintassi:

```
<animator attribute=”nomeAttributo”
    from=”value” to=”value” duration=”msec” />
```

- Se si vogliono **animare più attributi**:

  <animatorgroup process**="**.....**">**

  <animator attribute=”attributo_1” ..... />

  <animator attribute=”attributo_2” ..... />

  </animatorgroup>

- Con process si specifica come devono essere eseguite le singole animazioni:
  - *“simultaneous”*
  - *“sequential”*
- I tag *<animatorgroup>* possono essere **annidati** per animazioni più complesse

2° esercitazione

1. Inserire un componente a scelta e permetterne la modifica dello stato **visualizzato/nascosto** a seconda che una **checkbox** sia spuntata o meno
2. Creare una **finestra** ridimensionabile, ed al suo interno posizionare qualche componente le cui dimensioni/posizioni siano legate a quelle della finestra stessa
3. Creare delle animazioni di un componente a scelta che coinvolga più attributi

OpenLaszlo:

Come definire la LOGICA

Scripting

- LZX permette di programmare la logica dell'interfaccia per mezzo di un **linguaggio procedurale** basato su *JavaScript*
- Il codice viene eseguito sul **client**, *quindi attenzione ad un carico eccessivo*
- L'**interprete** del linguaggio è interno al Flash Player *(no JavaScript del browser)*, quindi verrà eseguito su ogni client

- Sintassi essenziale (simile a *C* e *Java*)
  - **Ogni "cosa" è un oggetto**
    (String, Number, Array, Boolean, Date, Function, Math, Object...)
  - **Strutture di controllo e condizionali**
    if (*...condizione...*) { *...istruzioni...* } **else** { *...istruzioni...* }
    while (*...condizione...*) { *...istruzioni...* }
    for (..... ; ..... ; .....) { *...istruzioni...* }
    switch (*...espressione...*) {
    case *CASE_1* : *...istruzioni...*; **break;**
    .....
    default : *...istruzioni...*;
    }
  - **Funzioni e procedure**
    function (*arg1, arg2, arg3*) {
    ...istruzioni...; **return** *...espressione...;*
    }

Attenzione ai caratteri illegali in XML!!!

```
<script>
    if ( this.x < 100 ) Debug.info( this.x );
</script>
```

Questo carattere va in contrasto con l'eventuale apertura di un nuovo tag!!!

Soluzioni:

- sostituire i caratteri incriminati con **entità HTML**

```
<script>
    if ( this.x  &lt;  100 ) Debug.info( this.x );
<script>
```

- inglobare il codice in una **sezione CDATA**

```
<script>
    <![CDATA[
    if ( this.x < 100 ) Debug.info( this.x );
    ]]>
<script>
```

- Per dettagli e differenze da *JavaScript standard* vedere nella guida:
  http://www.openlaszlo.org/lps4.2/docs/developers/ecmascript-and-lzx.html
- Le applicazioni OpenLaszlo possono include **codice procedurale**:
  - nei metodi di un oggetto (tag <method>)
  - negli eventi di un oggetto (tag <handler>)
  - nei tag <script>
  - nelle espressioni all'interno dei *constraints* per il calcolo del valore di un attributo

I tag <*script*>

- Il codice inserito all'interno dei tag <*script*> viene **eseguito immediatamente**
- Possono comparire solo nel <*canvas*>
- Utili per definizioni/impostazioni iniziali
- Esempio:

```
<script>
    Debug.write( myView.xoffset )
    myText.setAttribute("text", "hello world");
</script>
```

I tag <*method*>

- Il codice inserito dentro un tag *<method>* viene **eseguito quando viene richiamato** con la chiamata *nomeOggetto.nomeMetodo()*
- Sintassi:

  <method name=”nomeMetodo**” args=”**arg_1, …, arg_n**”>**
  …codice del metodo…
  </method>

  - Un metodo può avere **zero o più argomenti** in ingresso e
  - **può restituire un valore** in uscita con il comando **return** espressione**;**

- E’ possibile l’**overloading dei metodi**:
  - il metodo originale *(o della superclasse)* non viene richiamato una volta che viene ridefinito
  - per richiamarlo esplicitamente si usa la sintassi **super**.nomeMetodo(...)

- Esempi di metodi:
  - <view>
    getAttribute(), setAttribute()
    play(), stop(), seek(), setVolume(), unload(), destroy()
  - <animator>
    doStart(), pause(), stop()
  - <window>
    open(), close(), bringToFront(), sendToBack()

I tag <*handler*>

- Paradigma di **programmazione ad eventi:**
  - normalmente l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in ben determinati punti predefiniti dal programmatore *(if, switch ecc)*
  - nel paradigma ad eventi, invece, il flusso del programma è largamente determinato dal verificarsi di eventi esterni *(quando viene premuto il mouse, al termine del caricamento di una immagine ecc)*
- Il codice inserito dentro un tag *<handler>* viene **eseguito al verificarsi delle condizioni** che determinano quel particolare evento

- Sintassi:

  <handler name=”nomeEvento**”>**
  …codice per questo evento…
  </handler>

- Esempi di eventi:
  - <view>
    onclick, ondblclick, onmouseover, onmouseout
    onfocus, onkeydown, onkeyup
    oninit, onload, onerror, ontimeout
  - <animator>
    onstart, onstop, onpaused
  - <text>
    ontext

Eventi per la modifica degli attributi:

ogniqualvolta si modifica un attributo con il metodo setAttribute() viene generato un conseguente evento:

myView.**setAttribute('x',**this.x+10**)** → Evento **onx** per l'oggetto *myView*

myWin.**setAttribute('width',**400**)** → Evento **onwidth** per l'oggetto *myWin*

- Questi eventi stanno alla base del meccanismo interno dei *constraints*
- Per non scatenare i conseguenti eventi, basta modificare gli attributi direttamente:

myView.x **=** this.x +10; oppure myWin.width **=** 400;

Note su *error handling*

Quando si effettua una richiesta HTTP per un dato *(immagini, suoni o dati XML)* ci si deve sempre ricordare di **controllare l'esito della richiesta**:

- **onload**/**ondata** viene generato al termine della ricezione corretta del dato richiesto;
- **onerror** è lanciato al verificarsi di errori durante la trasmissione o se il formato del dato è errato;
- **ontimeout** viene generato dopo che non si ricevono dati per un periodo di tempo prestabilito (default 30s).

3° esercitazione

1. Creare una classe che implementi un **pulsante** che:
    - si ingrandisca al passaggio del mouse;
    - visualizzi icone differenti a seconda dello stato;
    - emetta dei suoni alle eventuali azioni del mouse.
2. Creare una **finestra** che:
    - non possa essere spostata oltre un box posizionato in alto;
    - sia pilotata da un pulsante;
    - si apra/chiuda con una animazione articolata;
    - il cui stato (aperto/chiuso) sia correttamente rappresentato dal pulsante.

3. Dentro la finestra, caricare una **immagine** a scelta tra quelle presenti in una **combobox**:
    - visualizzare correttamente l'immagine *(dimensioni, clipping, stretching);*
    - gestire eventuali errori nel caricamento;
    - visualizzare le informazioni dell'immagine (dimensioni, filename ecc).
4. Dare la possibilità di effettuare lo **zoom** ed il **pan** tramite 6 pulsanti grafici (es. )
5. Provare a caricare **più immagini** e gestirle in un layout, mantenendo l'aspect-ratio ed applicando i parametri di zoom/pan a tutte