AS3

trace("Action Script 3");

"Porque o AS3 pode ser entendido."

Escrito por Mário Santos
http://www.msdevstudio.com

INDEX

brevemente

1. Introdução

O que é o Action Script 3?

O action script 3 nasceu da necessidade de conseguir ter uma melhor performance nas aplicações e animações flash, actualmente sabe-se que teria necessariamente que ser implementadas alterações de raiz no action script 2 para que este pudesse ter a performance do actual AS3, por isso foi preferível quase que criar a linguagem de novo à parcial alteração do AS2. A linguagem AS3 é literalmente tratada à parte da AS2 pela runtime do flash player, pelo que logo aí se pode concluir que são mesmo diferentes a nível de código e de performance também.

O grande ponto forte do AS3 é que este é essencialmente focado em Programação Orientada a Objectos, pelo que muitos programadores sentirão dificuldades em compreender bastantes aspectos desta linguagem. Claro que os programadores Java partem em vantagem para a aprendizagem, mas para todos os outros fica uma dica. O AS3 não é nenhum bicho de 7 cabeças, e a sua vertente OOP torna as coisas muito simples, rápidas e organizadas. Para os programadores AS2 as coisas não serão muito fáceis, muito pelo facto de à primeira vista muitas coisas possam ser compreendidas e interpretadas como no AS2, mas posso vos dizer por experiência própria que depois de se aprender as bases, torna-se "quase mais" simples que o AS2.

Um dos grandes problemas que se apresentam ao programar em Action Script 3 é a impossibilidade de programar usando AS3 e AS2 mutuamente, porque como já foi dito em cima eles são tratados de forma diferente pelo Flash Player, logo se querem testar o AS3, terão obrigatoriamente de saber alguns pontos da linguagem.

Uma grande diferença do AS3 é que este exige obrigatoriamente/estritamente a declaração de variáveis, argumentos, funções e seus valores devolvidos, mesmo que a função não devolva nada ou o argumento nulo. Na versão anterior, AS2, esta declaração era opcional, pelo que uma simples função:

```
function olaMundo() {
}
```

funcionava exactamente igual a

```
function olaMundo():void {}
```

No AS3, o primeiro método foi eliminado, pelo que todas as funções, argumentos e valores devolvidos (return's) têm que ser devidamente declarados.

A nível sintáctico de linguagem, muitas coisas foram eliminadas, outras simplificadas e ainda outras melhoradas, tais como os muitos métodos de adicionar objectos à display list (main stage) foram agora simplificados e reduzidos a apenas 2 comandos para adicionar um "child" e remove-lo.

A nível de eventos, existem muitas melhorias no AS3, visto que agora quase todos os elementos gráficos podem ser sujeitos a "escutas" de eventos, separando por completo cada um o que permite uma maior autonomia bem como versatilidade.

Existem também melhoramentos ao nível do trabalho com XML que podem ser agora manipulados quase como objectos/items no AS, bem como o melhoramento dos elementos de texto, permitindo agora um melhor controlo sobre texto.

Foram também adicionadas bastantes melhorias no que diz respeito ao trabalho com sons, vídeo, dados de uma linguagem server-side e objectos em si.

O Action Script é directamente orientado para a plataforma Flash, pelo que está sempre dependente da Adobe, mas actualmente com a evolução do mudo das Aplicações Ricas para Internet, já começamos a assistir a uma tendência de extender esta linguagem para aplicações desktop, embora ainda baseadas na plataforma Flash, já apresenta um grau de independência enorme, isto tudo graças à plataforma AIR.

Uma clássica estrutura de Objectos e sua Linguagem Orientada, é que podemos ter um objecto Principal (Collection), e depois as suas categorias internas (Classes), que por si são um conjunto de código, vejamos com o exemplo:

Transportes como objecto principal, dentro dele podemos ter Carro, Avião, Moto e Autocarro e dentro de cada um destes temos os seus procedimentos, imaginemos:

Transportes -> Carro -> Levo 5 pessoas
-> Avião -> Levo 350 pessoas
-> Moto -> Levo 2 Pessoas
-> Autocarro -> Levo 50 Pessoas

O processo de dividir o código por pequenas instâncias chama-se "Encapsulation", que é uma das imagens de marca do AS3.

Poderemos ainda ter uma cadeia, no exemplo do carro:

Transporte -> Carro -> Bancos da frente -> 2 Pessoas
-> Bancos de trás -> 3 Pessoas

Este processo chama-se inheritance, que é o processo de podermos implementar/extender uma das classes base, neste caso extendemos a class Carro do Objecto Transporte.

Este tipo de organização pode ser definida pelo utilizador, não tendo qualquer tipo de limitação, apenas temos de ter em conta quando se implementa/extender uma class, já que estamos sujeitos às suas definições, ou seja, imaginemos no caso do carro, não podemos extender a class Carro da seguinte forma:

Transporte - > Carro -> Bancos da frente -> 2 Pessoas
-> Bancos de trás -> 5 Pessoas

Isto a titulo de exemplo não seria correcto e conduziria-nos a um erro, visto que o carro por definição apenas contém 5 lugares e objectivamente nunca deve levar mais de 5 pessoas. O que pretendo dizer com isto é que ao extendermos/utilizar-mos uma class como base devemos tem em atenção as suas definições e limitações.

O melhor destes aspectos do AS3 é que o código tem uma facilidade enorme de ser interpretado, bem como reutilizado e partilhado.

A nível do seu "core" o Action Script é baseado na linguagem ECMA-262 mais conhecida como ECMAScript que cresceu com o sucesso da sua linguagem mais conhecida, o Javascript. O AS2 foi estabilizado durante bastante tempo, mas a crescente necessidade de melhorar e acompanhar as tendências do mercado e programadores, o AS3 foi reescrito do nada, contando com uma enorme possibilidade de expansão e quase "infinitas" características que acompanham as muitas linguagens de programação, temos por exemplo actualmente o suporte nativo a objectos e cenas 3D no futuro Flash Player 10, isto praticamente só possível devido à migração do AS2 para o AS3 do seu motor. Esta grande alteração do AS3 acrescenta inúmero pontos novos, mas as suas bases mantêm-se intactas.

2.Bases da linguagem

Como todas as linguagens de programação e sua devida iniciação, vou passar a explicar as bases da linguagem.

2.1.Ordem de Execução:

De forma geral o AS3 é executado de Cima para Baixo e linearmente da esquerda para a direita:

passo 1 -> passo 2 -> passo 3
passo 4 -> passo 5
passo 6

Existem alguns argumentos que podem alterar esta ordem, mas as bases de execução são estas, esses elementos podem ser por exemplo a chamada de um passo diferente da ordem de execução, imaginemos:

passo 1 -> passo 2 (chama passo 6) -> passo 3
passo 4 -> passo 5
passo 6 (termina passo 6) -> passo 7

Neste caso a ordem seria
passo 1 -> passo 2 -> (passo 6) -> passo 3
passo 4 -> passo 5

Isto em si não é dificil de entender, o passo 2 chama o passo 6 e interrompe a execução normal, só depois de o passo 6 executado é que a execução prossegue para o passo 3.

2.2. Uso do ponto e virgula (semicolon) ";".

O uso oficial do ; serve para executar mais que uma instrução numa linha e também para indicar o final de uma linha de código. Esta indicação no final da linha, ao contrário do que se pensa não é obrigatória a sua inserção no final de cada linha, visto que não é disparado qualquer erro, mas por uma questão de organização e por facilidade de compreensão de outras linguagens.

2.3. Uso do Trace.

O trace como em muitas linguagens é muito util no caso de debug, para obtermos valores de determinadas variáveis e funções. Este trace() coloca no painel output do flash o valor que lhe indicar-mos, no caso de trace("olá Mundo") aparecerá no painel de debug olá Mundo. Notem que no caso do Flex, necessitam de ter o plugin com o debugger instalado, caso contrário não conseguirão fazer o devido debug.

2.4. Variáveis e tipos de dados.

As variáveis AS3, tais como as variáveis de outras linguagens de programação são usadas para guardar valores. As variáveis devem ser identificadas como únicas e que não interfiram com os elementos da própria linguagem em si.
Como por exemplo:

minhaVariavel1 = 1;
variav2 = 2;
Um erro:
var = 2;

Este erro acontece porque o var é explicitamente usado pela linguagem de programação para, curiosamente, declarar variàveis.

As regras base para declaração devariaveis são algumas; Não devem iniciar com números, devem apenas conter letras, o simbolo "$" ou o "_", não existir já uma variável com esse nome, nem o seu nome ser igual a uma variavel ou função protegida/reservada pelo AS3.

Para evitar estes erros, o próprio AS notificará se a variável já estiver em uso, ou se estiver protegida. Em ultimo caso o valor da variàvel será inválido.
Mas para que tudo isto funcione a variavel deve ser devidamente identificada e declarada:

var minhaVar1:Number = 1; e como erro teriamos:

var minhaVar1:Number = "1"; já que aqui estariamos a associar uma "string" a uma variável que apenas suporta tipo "Number", ou seja, numeros.

Existem vários tipos de declarações de variàveis, vejamos apenas as mais comuns e seus dados:

Number: Ex. 2.57, pode guardar numeros inteiros e décimas/milhares
int: Ex -7, Qualquer inteiro
uint: Ex. 5, Qualquer numero positivo.
String: Ex. "Olá", Qualquer cadeia de caracteres ou texto.
Boolean: Ex. true, Apenas verdadeiro (true) ou falso (false)
Array: Ex [1, 5, "sete"], Guarda mais que um valor numa variável, String ou Numero
Object: Ex. meuObjecto, Aqui uma das inumera possibilidades, podemos atribuir um objecto a uma variàvel, quer seja um campo de texto ou um movieClip.

Exemplo de uma variável Object.
var meuObjecto:Object = new MovieClip();
Declaramos a variavél meuObjecto como objecto e inicia-mo-la com um novo (new) MovieClip.
Na versão anterior do AS (2) esta declaração era opcional, mas no AS3 esta declaração é sempre obrigatória.

2.4.1. Casting (Variavel ou objecto)

O casting é uma maneira de definirmos uma variável / objecto como outro tipo de variável/Objecto.
Por exemplo, temos a variavel var a:String="1"; e temos uma variável numero var num:Number; e vamos supor, que por um motivo que seja, temos a variável a com diferentes valores numéricos:

a='1';
a='1500';
e por ai alem...

Se fizermos num=a; vamos cometer um erro, para isso usamos o casting, que serve para obrigar essa variável a ser identificada como outra:

num=Number(a);
e o num passará agora a ser considerado o valor numérico da string a, de notar que se fizerem:

a="1 unidade"; e
num=Number(a);

não vai funcionar, porque o casting é incapaz de transformar o "unidade" em valor numérico, mas mais à frente falaremos com mais atenção sobre as propriedades e possibilidades do casting.

2.5. Condições e Operadores:

2.5.1. IF - if ()

Este é o mais comuns dos operadores de condições, verifica a comparação entre determinados elementos, mais comuns em comparação de variáveis.
Ex.

```
var a:Number = 1;
var b:String= "ola";
var c:Boolean = false;

if (a == 1) {
trace("a=1 é verdadeiro");
}
```

O uso do operador == é bastante importante, porque faz a comparação sem associação, ou sea, compara o valor absoluto. Alguns tipos de operadores são o > (maior que), < (menor que), >= (maior ou igual que), <= (menor ou igual que) bem como muitos outros. Dentro de um if podemos fazer várias comparações ao mesmo tempo, sendo que todas são verificadas dentro d mesmo if, e basta uma dessas condições não ser verdadeira (no caso de usarmos apenas o &&) para que invalide a comparação. Temos outros operadores, como o OR (||) ou o NOT (!) junto com o AND (&&).

Como exemplo a retornar falso o seguinte if (as 2 condições tem que se verificar):

```
if(a == 1 && b =="mundo" ) {
trace("a=1 e b= mundo");
}
```

O mesmo exemplo mas a retornar verdadeiro (apenas uma das condições tem que se verificar)

```
if(a == 1 || b =="mundo" ) {
trace("a=1 e b= mundo");
}
```

Regra geral o if testa sempre a "verdade", ou seja:

if (!c)

vai retornar verdadeiro, já que verifica se o c = false.

O operador ! pode também ser usado para complementar o igual (=), como por exemplo verificar se o a é diferente do numero 1:

if(a != 2) é verdade.

Junto com o if, podemos usar a o "else" para detectar a invalidação da comparação:

```
if( a != 2) {
trace("Sim, o a é diferente de 2")
}
else {
trace("O a é mesmo igual a 2");
}
```

E ainda dentro do else, podemos testar uma nova condição id, com o elseif();

```
if( a == 2) {
trace("Sim, o a é igual a 2")
}
elseif( a == 1) {
trace("O a é mesmo igual a 1");
}
else {
trace("O a não é igual a 2 nem a 1");
}
```

2.5.2. SWITCH - switch();

O switch passa por ser uma alternativa mais simples de compreender quando ser tornam necessárias várias comparações, visto que o if no caos de 5 comparações individuais seria muito extenso. Imaginemos no caso em cima, que queríamos verificar o a umas 5 vezes, teríamos 5 if's e elseif's, mas com o switch apenas o seguinte:

```
switch (a) {
    case 1:
    trace("o a é igual a 1");
    break;
    case 2:
    trace("o a é igual a 2");
    break;
    case 3:
    trace("o a é igual a 3");
    break;
    case 4:
    trace("o a é igual a 4");
    break;
    case 5:
    trace("o a é igual a 5");
    break;
    default:
    trace("o a não é ugual a nenhum dos valore acima.");
    break;
 }
```

Em muitos casos o switch passa por ser a melhor opção. Os break; são essenciais ao funcionamento do switch, senão os case serão todos comparados. Apenas o ultimo case pode ser dispensado, visto que não existe mais condição nenhuma a seguir.

2.6. Ciclos.

Usamos os cliclos quando necessitamos de fazer uma repetição de determinada condição ou acção.

2.6.1. Ciclo FOR for()

Composto por 3 argumentos, o que, de onde, até onde. Este ciclo é muito útil para fazer comparações ou execuções repetidamente um determinado numero de vezes, como por exemplo:

```
for (var i:Number = 0; i < 15; i++) {
 trace("Olá numero "+i);
 }
```

Este ciclo é executado 15 vezes, contando com o i=0 até ao i<15 (i=14);

Um outro Exemplo:

```
for (var i:Number = 15; i > 0; i--) {
trace("Olá numero"+i);
}
```

Este ciclo também é executado 15 vezes, mas da forma contraria, cada vez que é executado o i diminui até chegar a i=1;

2.6.2. Ciclo WHILE while()

Este é outro tipo de ciclo, que se executa enquanto a condição for verdadeira, tomemos como exemplo:

```
var num:Number = 0;
while (num < 5) {
num++;
}
```

Neste exemplo declaramos o num como Number com o valor 0, e o ciclo é efectuado enquanto essa variável num for menor que cinco, ou seja, o ciclo é executado 5 vezes visto que a variável num está a ser incrementada numero a numero, ou num++; é a mesma coisa que num=num+1;

Deve-se ter muita atenção a usar loops (for e while) visto que estes loops utilizam muitos recursos do flash, e durante a sua execução praticamente nada será feito, ou seja, o loop interrompe o processo normal do action script e só liberta os recursos assim que termina. Ou seja, no loops em cima, principalmente no while, a ausência do num++ faria com que o ciclo entrasse em execução infinita, pelo que a aplicação / animação pararia literalmente e muito provavelmente até o browser irá parar.

Devemos ter ainda mais especial atenção quando temos alguns ou bastantes elementos gráficos a apresentar ao utilizador e principalmente não executar ciclos que possam ser muito complexos durante a construção da display list, e se for mesmo necessário devemos executa-los por ordem e nunca ao mesmo tempo que a apresentação de elementos críticos no stage.

O Stage em si é a area que servirá de portador/"container" para todos os elementos gráficos, caso eles existam.

2.7. Arrays

Enquanto as variáveis "normais" apenas suportam um tipo de dados e um valor, um array consegue suportar varios valores. Num exemplo prático, suponhamos que necessitamos de guardar 50 nomes de pessoas, não vamos declarar 50 variáveis visto que o processamento do action script se tornaria mais lento, além do nosso código ficar enorme.
Como foi dito, podemos guardar num array varios valores, fazendo a declaração da seguinte forma.

```
var meuArray1:Array = [1, 2, 3];
var meuArray2:Array = new Array();
```

Ambos os arrays podem ser facilmente modificados, vejamos:

Como adicionar um elemento ao nosso array:
```
meuArray2.push(1);
```
E o valor 1 é adicionado ao array;

Como remover o ultimo item ao array:
```
meuArray2.pop();
```

Como buscar o valor de determinada posição dentro do array:
```
var meuArray3:Array = ["1"," 2"," 3"];
```

trace(meuArray[0]); irá retornar a string "1"

Existem muitas mais operações relativas a arrays, como os percorrer, comparar, eliminar um item em certa posição, procurar, etc... Os arrays têm a possibilidade de serem também arrays matriciais ou multi-dimensionais e receber varias variáveis em varias posições, bem como receber outro array em cada posição que falaremos mais à frente...

3. Objectos (Object)

Depois de alguma experiencia com Actioin Script, facilmente se apercebem que muitas propriedades de Objectos, por exemplo um MovieClip, são acedidas e disponibilizadas propriedades através de um acesso rápido, como por exemplo:

```
var meuMovieClip:MovieClip;

meuMovieClip.x=posição_eixo_xx;
meuMovieClip.y=posição_eixo_yy;
meuMovieClip.width=tamanho_comprimento;
```

e muitas mais propriedades, isto se chama um objecto e desde já ficamos a saber que grande parte de componentes actionScript são derivados de Objectos.

Como o AS3 é bastante versátil, permite-nos também criar o nosso objecto personalizado e definir-mos as características que quisermos... vejamos:

```
var objectoTeste:Object = new Object();

objectoTeste.param1=5;
objectoTeste.param2=false;
objectoTeste.param3=null;
```

Como podemos ver pelo exemplo, criámos as nossas próprias propriedades do nosso objecto sem dificuldade nenhuma, e para lhes aceder é tão simples como:

```
var zed:Boolean;
zed=objectoTeste.param2;
trace("o valor do zed é "+zed);
```

Alem de servir para criarmos os nossos proprios objectos, pode servir também para um rápido acesso a um objecto com muitas caracteristicas, e principalmente torna-se muito útil para enviar um conjunto de dados para uma função em vez de utilizarmos um array, como no exemplo que se segue usando o objecto em cima:

```
function mostraEstadoObjecto(obj:Object):void {
trace(obj.param1);
trace(obj.param2);
trace(obj.param3);
};

mostraEstadoObjecto(objectoTeste);
```

Serão então apresentados os valores das devidas propriedades do nosso objectoTeste; Notem que o obj recebido na função passa a ser considerado como o nosso objectoTeste e por isso com as mesmas propriedades.

4. Funções

As funções são uma parte indispensavel para os programadores em geral, e no AS3 são também uma parte crucial. Sem funções, o código seria executado conforme indicado anteriormente; sequencial de cima para baixo e da esquerda para a direita, com as funções, o código dentro delas apenas é executado quando elas são chamadas, e apenas indexadas pelo runtime.

As funções tornam o nosso código mais limpo, evitando que sejam executadas instruções e processamentos sem necessidade nessa altura. Para criar uma função, temos alguns passos simples a seguir, pricipalmente porque aqui reside uma grande diferença do AS2 que é a necessidade de declarar o seu tipo, retorno e possiveis argumentos.

Temos como exemplo a seguinte função simples:

```
private function mostraMsg(){
trace("olá");
}
```

Que pare ser executada deve ser chamada com o comendo a seguir:

```
mostraMsg();
```

Esta função em si não está bem declarada, mesmo não devolvendo nada deve-se indicar ao compilador que ela não devolve mesmo nada, ficaria então assim:

```
private function mostraMsg():void {
trace("olá");
}
```

As funções devem também ser indicadas com o tipo de função que é, neste caso ou private ou public sendo que a grande diferença entre elas é que conforme o nome diz, uma função provada, que não pode ser acedida fora do seu contexto ou do seu componente o que pelo contrario a função public permite o seu acesso fora desse contexto / componente.

No caso desta função, e de muitas outras, por uma questão de reutilização de código podemos simplificar as coisas e passar um ou mais parametros para a função. Isto torna-se muito util no caso de querer-mos fazer uma operação com um valor diferente, usando a função em cima e o que já foi explicado poderimaos fazer a função:

```
private function mostraMsg(msg:String):void {
trace(msg);
}
```

e chama-la como mostraMsg("olá");

Esta operação apenas fará o mesmo que a anterior, mas de uma maneira reutilizável... podemos ainda tomar como por exemplo o que já foi dito em cima, para criarmos a nossa primeira função conjugando quase tudo o que foi dito em cima:

```
var dados:Array = ["olá","mundo","sou","um","teste"];

private function mostraMsg(msg:String):void {
trace(msg);
}

private function teste ():void {

for (var i:Number = 0; i < dados.length; i++) {
   mostraMsg(dados[i]);
   }

}

teste();
```

Ao executar-mos esta função teste(), será executado um ciclo de i=0 até i=4 (o dados.length retorna o tamanho do array) e m cada posição será chamada a função mostraMsg com a string da posição do array, ou seja, a função mostraMsg será chamada 5 vezes:

```
mostraMsg("olá");
mostraMsg("mundo");
mostraMsg("sou");
mostraMsg("um");
mostraMsg("teste");
```

o que produzirá um trace a estes cinco elementos do array.

As funções em si podem devolver dados... no caso em cima não devolvemos nada (:void), mas podemos devolver praticamente qualquer tipo de dados, ora vejamos:

```
private function percentagem(num:Number):Number {
return ((num/100));
}
```

Neste caso ao chamar-mos a função percentagem, ser-nos-à devolvido o valor percentual do numero que indicamos:

```
var perc:Number;
perc=percentagem(25);
```

a variavel perc ficará com o valor de (25/100) = 0,25

Com falamos antes em bases da linguagem, já pouco mais há a dizer, visto que com o decorrer do tutorial falaremos mais antentamente sobre algumas propriedades da linguagem...

Vamos agora passar a falar do "core" do Action Script 3.

3. Propriedades, métodos e eventos

Depois de já termos falado das bases do AS3, vamos falar agora dos elementos base para poder construir, definir e instruir muitos dos elementos graficos no Flex/Flash.

As propriedades são algo que define, uma cacateristica e em ultimo caso chamada de parâmetro. Como na vida real podemos definir um carro como marca, colorido, portas , tipo ou cilindrada, no action script podemos fazer exactamente o mesmo, mas no caso de um carro, apenas teríamos propriedades fixas, que não podemos alterar facilmente (a não ser na oficina), enquanto que no action script e seus objectos as suas propriedades são maioritariamente de leitura-escrita, ou seja, num botão por exemplo, podemos obter o seu tamanho pela propriedade meuBotao.width, ao mesmo tempo também podemos definir o seu tamanho com meuBotao.width=250; Existem também algumas propriedades que não podem ser alteradas (leitura apenas) mas como no caso do carro, também podemos altera-las, não muito facilmente nem directamente, mas mais para o final do tutorial explicarei com o fazer.

Os métodos são geralmente opções que permitem ao programador fazer com que um objecto execute certas operações, na maioria, internas que podem tanto definir varias propriedades do

objecto, ou efectuar uma simples acção. Por exemplo tendo um método definePosição() podemos definir o valor do x e do y em apenas uma linha de código, simplificando as definições das propriedades .x e .y. Em caso unico, pode ser usados para especificar uma acção única, como por exemplo o navigateToURL() que abre determinado site no browser .

Eventos são retornos de determinadas acções, por exemplo num carro, ao desligar o motor é disparado um imaginário evento "stop", no AS3, por exemplo um botão, se o utilizador clicar com o rato no botão é então acionado o evento click, e esse evento pode ser interceptado de maneira a que possamos efectuar uma acçao, regra geral uma função. O "dispositivo" que lida com estes eventos, chama-se "Event Handler" e é a partir deste handler que são chamadas as possíveis funções. No action script 3 foram implantados já certas funções de Event Handling, que são chamados os Event Listners, em tradução à letra, são "escutadores de eventos" que servem precisamente para detectar quando determinado evento foi "disparado", ou seja, quando o evento foi accionado, e a partir dessa detecção podemos saber em que preciso momento o evento foi disparado.
Fora estes eventos já implantados pelo AS3, podemos ainda criar os nossos próprios eventos, mas para isso teremos que explicar melhor como eles funcionam, o que acontecerá mais à frente.

3.1. Propriedades

No caso do action script, por exemplo em Flex, temos um painel, e dentro dele temos um botão... ambos os elementos partilham das mesmas características (independentes), como por exemplo o .x e o .y, mas se alterar-mos o .x do painel, o .x do botão não vai alterar, porque ele está dependente do painel, embora a posição absoluta no ecrã dele vá variar, a sua posição relativa ao painel não varia, a isto chama-se Child's, ou seja, filhos, no caso de definir-mos as propriedades do botão, por exemplo o .x e .y a sua posição será sempre calculada relativamente ao painel, e no caso de definirmos o tamanho do botão (.width e .height) maior que o painel, o botão terá gráficamente o tamanho máximo igual ao do painel e nunca superior, porque depende do tamanho do seu pai, ou seja, do seu "container" (portador/parent).

Regra geral, todos os elementos gráficos e não só são definidos usando propriedades que servem perfeitamente para podermos "personalizar" esse elemento, muito deles como no caso do flex aceitam uma personalização extrema muito graças ao suporte de skin's e css.
Alguns exemplos de propriedades:

```
meuBotão.label="texto"
meuBotão.x=200;
meuBotão.width=150;
meuBotão.visible=false;

meuMovieClip.alpha=0.5;
meuMovieClip.visible=true;
meuMovieClip.x=125;
```

Como podem ver, estes elementos tem propriedades partilhadas, como o caso do x, width, height, y e mais algumas outras...pelo que ao compreender-mos as propriedades principais, facilmente conseguiremos lidar com outros objectos. Algumas propriedades podem ser definidas com percentagens, imaginemos 50%, em action script definimos como 0.5, podem definir como percentagem algumas propriedades como width, height, alpha bem como muitas outras propriedades.

Nota:
Todos os elementos gráficos são filhos (child's) de algum outro, no caso de adicionar-mos apenas um painel na nossa aplicação, este será sempre filho do "stage " (palco/aplicação) principal.

3.2. Eventos

A maioria dos elementos gráficos e não só do Flash, Flex dependem em muito dos eventos, para termos noção, desde um simples click com o rato num botão, imagem ou texto, passando pela escrita de texto numa caixa de texto até um efeito, tudo dispara eventos, alguns deles muito importantes para a versatilidade do AS3.
Existe uma enorme variedade de eventos, tanto em objectos gráficos, em classes internas e até em classes desenvolvidas por terceiros, pelo que se torna muito importante ter uma boa noção dos mesmos visto que o AS3 é literalmente orientado a objectos é essencial conhecermos os eventos disparados por esse objecto para termos um melhor controlo sobre a nossa aplicação/animação.

No anterior AS2, bastantes eventos poderiam ser "interceptados" directamente como por exemplo com o on(release), on(press) ou até on(keyPress), mas como a linguagem evoluiu, actualmente no AS3 as coisas mudaram um pouco de figura, passando agora a ser directamente acedidas no objecto, tal como meuObjecto.onRelease, mas alguns destes eventos estão limitados no seu conteudo, e aí é que entram os "eventListner's" que podem ser aplicados a qualquer objecto, instancia ou elemento, pelo que não só nos dão a possibilidade de um melhor controlo, como a infinita possibilidade de criar, disparar e lidar com eventos pessoais, ou seja, eventos criados por nós mesmos.

O "eventDispatcher" (traduzido à letra, disparador de eventos) não é novo no AS3, mas foi muito melhorado no AS3 e incrementado passando a ser responsável pela maioria dos eventos que além de controlar os eventos por defeito das instancias, passa a controlar também os eventos personalizados permitindo ao programador saber quando determinada acção foi efectuado permitindo um controlo absoluto sobre o nosso código.

3.2.1. Como usar eventListners?

O conceito dos eventListners é bem simples, passando pela simples indicação no código que a instância deve escutar determinado evento e "avisar" quando ele acontecer para que possamos executar determinada função.
Existe um pormenor que temos que ter em conta, certos eventos não são suportados por certas instâncias, por exemplo, um evento keyPress nunca será disparado numa imagem. O que quero dizer, é que como foi dito anteriormente, aqui também os eventos dependem das instâncias. Por exemplo, queremos que o utilizador ao clicar com o rato num botão seja

executada uma função (funcao_a_executar) que deve ser chamado sem os () (este é dos poucos casos onde a função pode (obrigatoriamente) ser chamada sem os ()).

```
meuBotao1.addEventListner(MouseEvent.CLICK, funcao_a_executar);

private function funcao_a_executar(event:MouseEvent):void {
trace("botão clickado")
}
```

Temos que notar 1 ponto muito importante, como a funcao_a_executar() é disparada a partir de um MouseEvent, ela obrigatoriamente receberá parâmetros e características desse evento (algumas informações sobre a instancia) logo por isso terá que receber como parâmetro o "pai" desse evento, neste caso o "pai" é o MouseEvent. Um argumento muito útil passado por esse evento é o target ou currentTarget que nos indica a instância que disparou o evento à qual podemos aceder mesmo sem saber o nome dessa instância.

Um exemplo bastante simples:
Temos 2 botões para aumentar/diminuir o tamanho de uma imagem 5 pixeis, iria-mos fazer assim:

```
botao_aumenta.addEventListner(MouseEvent.CLICK, aumenta);
botao_diminui.addEventListner(MouseEvent.CLICK, diminui);

private function aumenta(evt:MouseEvent):void {
        //para saber o nome da instancia:
        trace("instancia : "+evt.target);
        imagem1.width +=5;
}
private function diminui(evt:MouseEvent):void {
        //para saber o nome da instancia:
        trace("instancia : "+evt.target);
        imagem1.width -=5;
}
```

O símbolo += ou -= serve para juntar ou diminuir os valores de certa variável/propriedade.
Podemos associar vários eventListners ao mesmo objecto:

```
meuBotao1.addEventListner(MouseEvent.CLICK, botao_click);
meuBotao1.addEventListner(MouseEvent.MOUSE_OVER, botao_rato_em_cima);
meuBotao1.addEventListner(MouseEvent.MOUSE_OUT, botao_rato_fora);
```

e todas as funções receberiam o mesmo tipo de evento:

```
private function botao_click (evt:MouseEvent):void {}
private function botao_rato_em_cima (evt:MouseEvent):void{}
private function botao_rato_fora (evt:MouseEvent):void{}
```

finalmente, podemos evitar o uso de 3 funções, e reutilizar o código usando os parâmetros

passados pelo evento, para saber que tipo de evento foi disparado (se todos forem "filhos" do mesmo evento (MouseEvent)):

```
meuBotao1.addEventListner(MouseEvent.CLICK, botao_evento);
meuBotao1.addEventListner(MouseEvent.MOUSE_OVER, botao_evento);
meuBotao1.addEventListner(MouseEvent.MOUSE_OUT, botao_evento);
```

e a função:

```
private function botao_evento(evt:MouseEvent):void {

/*usando um switch para saber que tipo de evento foi disparado, acedendo ao
parametro type do evt */
switch (evt.type) {

        case MouseEvent.CLICK:
        trace("evento CLICK");
        break;

        case MouseEvent.MOUSE_OVER:
        trace("evento MOUSE OVER");
        break;

        case MouseEvent.MOUSE_OUT:
        trace("evento MOUSE_OUT");
        break;
}
}
```

Como podem ver, podemos poupar algumas linhas de código e optimizar o resultado final do swf usando a função botao_evento para lidar com qualquer tipo de eventos do rato (MouseEvent).

Podemos também usar eventos para chamar metodos de outros objectos, sempre através de uma função, usando a função em cima:

```
        case MouseEvent.CLICK:
        meuBotao1.move(btn.x+5, btn.y+5);
        break;
```

Neste caso estamos a chamar o método move() do botão, ou seja a dizer ao botão que se deve mover 5px para a direita e 5px para baixo. Isto será executado cada vez que se clicar no botão. Por final, podemos ainda fazer com o o botão dispare outro evento, entrando já nas propriedades internas do botão e forçan-do-o a disparar um evento interno, usando a função em cima, quero que o utilizador ao passar o rato em cima do botão, que seja disparado o evento click do botão:

```
        case MouseEvent.MOUSE_OVER:
        meuBotao1.dispatchEvent(new MouseEvent(MouseEvent.click));
        break;
```

com a linha em cima estou a dizer à minha instancia/botão (meuBotao1) para que dispare o evento CLICK, para isso temos que o forçar a disparar um novo evento, exactamente igual ao que o eventListner está à espera (meuBotão1.addEventListner(***MouseEvent.CLICK***)) indicando também o tipo de evento (***MouseEvent***), ou caso contrario ele disparará o evento, mas nada acontecerá; meuBotao1.dispatchEvent(new ***MouseEvent***(***MouseEvent.click***)); Se mantivermos toda a nossa função em cima com o switch (retirando o MOUSE_OUT):

```
private function botao_evento(evt:MouseEvent):void {

switch (evt.type) {

        case MouseEvent.CLICK:
        meuBotao1.move(btn.x+5, btn.y+5);
        break;

        case MouseEvent.MOUSE_OVER:
        meuBotao1.dispatchEvent(new MouseEvent(MouseEvent.click));
        break;

}
}
```

Acontecerá que caso o utilizador passe o rato sobre ou clique no meuBotao1 será interpretado como se tivesse clicado, já que será disparado o evento CLICK na mesma em ambos os casos, o que fará com que o meuBotao1 chame o método .move();

3.2.2. Como remover EventListners ?

Os eventListner's são muito uteis como puderam ver em cima, mas ao mesmo tempo podem ser causadores de problemas, bem como sobrecarregar o .swf final, vejamos no caso de querer-mos apenas que o botão apenas dispare o evento click uma única vez. No caso explicado anteriormente o botão disparará sempre o evento click e mouse over durante todo o tempo que o .swf tiver aberto, logo, se o nosso objectivo é apenas saber a primeira vez que o botão foi clicado, deixando o código em cima produziria um objectivo inválido, já que seria executado sempre que o user clica-se no botão. Aqui é que entram os removeListner's, ou seja, se pretendemos que o botão apenas dispare uma unica vez a função associada ao evento click teremos que remover esse eventListner assim que o nosso objectivo esteja concluído (apenas chamar a função uma unica vez) e para isso fariamos o seguinte, usando a função anteriormente:

```
meuBotao1.addEventListner(MouseEvent.CLICK, botao_click);
```

```
private function botao_evento(evt:MouseEvent):void {

switch (evt.type) {

        case MouseEvent.CLICK:
        meuBotao1.move(btn.x+5, btn.y+5);
        //temos o objectivo terminado, logo vamos remover o eventListner:
        meuBotao1.removeEventListner(MouseEvent.CLICK, botao_click);

        break;

}
```

Desta feita, a função botao_evento apenas será chamada pelo eventListner uma unica vez, já que o removemos (meuBotão1.removeEventListner(MouseEvent.CLICK, botao_click);), neste removeEventListner temos apenas que cumprir uma regra; o evento a ser removido terá que ser exactamente igual (**MouseEvent.CLICK**) ao adicionado (addEventListner) incluindo a função a ele associada (**botao_click**), já que podem existir várias funções associadas ao mesmo eventListner:

meuBotão1.removeEventListner(**MouseEvent.CLICK**, ***botao_click***);

3.2.3. Garbage Collector

Como o nome diz, traduzido á letra, é um reconhedor de lixo, é um mecanismo que o flash player possui para, durante a execução, inspecionar o swf e remover items e objectos não utilizados ou deixados esquecidos pelo programador. Regra geral os eventos não são removidos nessa recolha pelo garbage colector, porque o programador deve ter o bom senso de os remover, mas se for iniciante, provavelmente esquecerá um ou outro event listner, e por isso uma boa prática será marcar os seus event Listners para serem inspeccionados pelo garbage collector e removidos se desnecessários, para para isso quando criam o vosso event listner devem ter em atenção mais alguns parametros (sendo que apenas um deles é o mais importante).
Estou a falar de um argumento (useWeakReference) que indica ao garbage collector para o inspeccionar e remover se não for preciso, este argumento é o numero 5 na lista de argumentos dos addEventListner's:

addEventListner(type:String, listner:Function, **useCapture:Boolean=false, priority:int=0, useWeakReference:Boolean=false**)

Este 3 parâmetros a mais escuro são opcionais, explicando:
useCapture: Serve para "observar" o "caminho do evento" até este ser disparado no seu "target" objecto. Por defeito a **false**, e devem deixar estar assim, pelo menos quase sempre.
priority: Usado para definir a prioridade desse evento, no caso de existirem muitos eventos associados ao objecto. Por defeito a **0**, quase sempre usado assim.
useWeakReference: O mais importante dos 3, este por uma questão de boas práticas devem utilizar como **true** visto ser este parâmetro que indica ao garbage collector para o remover

caso ele seja esquecido. O uso desta opção não é uma alternativa ao removeEventListner e nem deve ser usado como tal, (usem sempre o removeEventListner), mas sim deve ser usado como uma questão de boas práticas, pelo que anteriormente nos nossos event listners deveremos usar:

meuBotão1.addEventListner(MouseEvent.CLICK, botao_evento, false, 0, **true**);
meuBotão1.addEventListner(MouseEvent.MOUSE_OVER, botao_evento, false, 0, **true**);
meuBotão1.addEventListner(MouseEvent.MOUSE_OUT, botao_evento, false, 0, **true**);

Se começarem a ganhar este hábito, dificilmente terão problemas de memória com os eventListner's, mesmo que se esqueçam de os remover, porque afinal de contas, muito mal se fala do sistema de limpeza de lixo do flash player, mas muitos desses problemas provem de maus hábitos de programação.

4. Display List

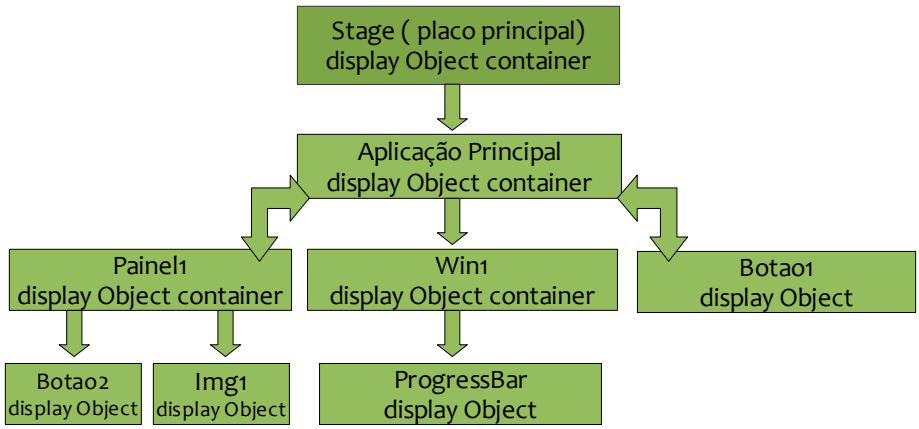
A display list, mais uma vez traduzida à letra pode ser entendida como lista de apresentação, foi uma grande inovação no no flash/flex com o AS3, e que agora apresentou muitos problemas de adaptação para muitos designers/programadores habituados à versão anterior. Nas versões anteriores do AS, todos os elementos visuais eram tratados e apresentados separadamente no swf final o que requesitava dos programadores um grande conhecimento e acima de tudo muito código para lidar com essa apresentação visual dos elementos e com os métodos de criar, eliminar e adicionar objectos o que resultava num controlo restrito desses mesmos objectos e que o garbage collector era incapaz de controlar.
O AS3 trouxe uma nova forma de lidar com estes objectos e sua apresentação, a display list. Logicamente esta display list pode ser entendida como uma lista de elementos visuais do swf que pode incluir os objectos mais comuns como movieclips e botões, mas também shapes, gráficos, painéis e muitos outros que não existiam nas versões anteriores ou que não podiam ser criados "programaticamente".
Resumindo, a display list é praticamente o que contém e regula a apresentação de todos os nossos elementos visuais na altura da "criação"/apresentação do .swf final.

4.1. Entendendo a display list.

Para compreender algo que não vemos, temos que ter provas bem lógicas do que se fala, e como disse em cima, a display list é o que gere todos os elementos gráficos do nosso .swf, falando no caso do flex, vou tentar fazer um rascunho do que podemos entender como sendo um esquema visual da display list. Suponhamos que em flex temos:

- Fundo da aplicação.
- 1 painel (Painel1)
- 1 titleWindow (Win1)
- 1 botão (botao1)
- 1 imagem dentro do painel 1 (Img1)
- 1 botão dentro do painel 1 (Botao2)
- 1 progressBar dentro do titleWindow (ProgressBar)

O esquema da display list apresentaria-se da seguinte forma:



Este é o exemplo de uma simples aplicação, pelo que a display list está organizada com vários objectos, os que enumerei como "display Object Container" são objecto que suportam outros objectos dentro de si como filhos ("child's"), e os enumerados com "display Object", não suportam "nativamente" filhos/"childs" dentro de si.
Como podem ver com este esquema, facilmente se conseguem perceber as vantagens da displayList, ou seja, na imagem conseguimos perceber alguns pontos:

- O stage é o palco principal, podemos entender como o Objecto que contém todos os outros, sendo que é o Pai ("root/raiz")
- A aplicação principal, que contém os elementos painel 1, win 1, botão 1 como seus child's e tendo como seu pai o stage (parent)
- Por sua vez, os child's da aplicação principal (painel 1 e win1) também contêm child's tendo como parent a applicação principal.
- Finalmente temos os objectos (botão2 e Img1) como childs do Painel 1 e o objecto ProgressBar como child do objecto Win1.

Isto pode parecer um pouco complicado, mas podem com uma simples função conseguimos perceber esta hierarquia de Objectos:

```
private function mostraFilhos(dispObj:DisplayObject):void {
        for (var i:int = 0; i< dispObj.numChildren; i++) {
                var obj:DisplayObject = dispObj.getChildAt(i);
                if (obj is DisplayObjectContainer) {
                        trace(obj.name, obj);
                        mostraFilhos(obj);
                }
                else
                 {
                trace(obj);
                }
        }
}
```

```
mostraFilhos(this);
```

Serão então apresentados no painel de debug os objectos e seus childs (se existirem).
Nesta função temos alguns parametros:
.numChildren; Mostra o numero de child's do objecto.
.getChildAt(i); Busca o nome do child na devida posição da diplayList;
DisplayObjectContainer; Compara se o objecto é um displayContainer, caso seja lista os seus filhos tambem, chamando a função de novo.

4.2. Adicionado um objecto à display List. addChild();

No flex, ao arrastar-mos um componente para dentro de um painel ou para a nossa area (fundo/Aplicação), na realidade mesmo vendo o código MXML gerado, não vemos nenhuma referencia ao addChild, pelo simples motivo; O MXML é uma linguagem estilo XML que serve unicamente para organizar o código de uma maneira mais simples e lógica, mas na realidade este MXML é completamente transformado em AS3 na altura da compilação do vosso projecto, e para que tenham a certeza disso saibam que não existe nada no MXML que não possamos fazer em AS3, vejam o exemplo:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Array id="simplesArray">
    <mx:String>texto1</mx:String>
    <mx:String>texto2</mx:String>
  </mx:Array>
  <mx:Panel title="painel1">
    <mx:ComboBox dataProvider="{simplesArray}"></mx:ComboBox>
  </mx:Panel>
</mx:Application>
```

Em action script 3 no flex isto ficaria algo como:

```
import mx.controls.ComboBox;
import mx.containers.Panel;

private var simplesArray:Array = new Array("texto1","texto2");

private var painel:Panel = new Panel();
private var combo:ComboBox = new ComboBox();

painel.id="painel1";
combo.dataProvider=simplesArray;
painel.addChild(combo);

addChild(painel);
```

Se reparar-mos poucas coisas temos que fazer, e o numero de linhas é quase o mesmo. Reparem com atenção nas propriedades e métodos que indiquei no código, penso que a única coisa nova será a propriedade **.id** que define o id do painel, e os imports no topo, que importam os objectos ComboBox e Panel para que possam ser usados. Podemos ver também os métodos addChild em acção, adicionando em primeiro o comboBox (combo) como child do painel e depois adicionando o painel ao stage principal.
O addChild é um método disponivel em muitos dos elementos gráficos presentes no flex, pelo que este exemplo pode ser aplicado a inumeros objectos, que tem como principal objectivo adicionar elementos à display list ou a outro elemento (display object container) da display list.

Sendo que para adicionar à display list, basta usar:

```
addChild(Objecto);
```

onde o objecto pode ser praticamente qualquer componente/displayObject do flex/flash

E para adicionar a outro objecto já filho/child da display list:

```
displayObjectContainer.addChild(Objecto);
```

onde o displayObjectContainer é o objecto ao qual será adicionado o child, no caso em cima, usamos o painel como display object container, e o combo como Objecto.

Podemos também adicionar um display object container (painel por exemplo) a outro display object Container (TitleWindow):

```
meuPainel.addChild(minhaTitleWin);
```

4.3. Removendo objectos da display list. removeChild ():

O processo para remover child's do stage ou de algum elemento é bem simples e muito parecido com o processo de os adicionar. O método removeChild está também presente em muitos dos elementos gráficos, pelo que usamos apenas:
removeChild(painel) e eleminaria-mos o child painel bem como todo o seu conteúdo que inclui o nosso objecto comboBox. Alem deste processo podemos usar também o método removeChildAt() que recebe como parametro o numero/posição do (child) objecto em causa, numero esse que é atribuido automaticamente pela display list, no exemplo em cima, o comboBox será o child numero o, visto que é o unico child adicionado ao painel, para o remover-mos devemos usar:

painel.removeChildAt(0); e o comboBox seria automaticamente removido do painel.
Vamos supor que na função em cima, adicionamos também um TextInput ao painel:

```
private var texto:TextInput = new TextInput();
```

e em baixo adicionaríamos ao painel:

```
painel.addChild(combo);
painel.addChild(texto);
```

Aqui o texto passaria a ser o child numero 1, e para o remover teríamos que usar:
painel.removeChild(texto) ou painel.removeChildAt(1);

Como podem ver o processo é bastante simples, mas na realidade, o objecto foi só removido da displayList, mas continua residente na memória, fazendo com que o garbage collector o ignore na sua inspecção, mais uma vez se deve adoptar boas práticas, e para realmente remover estes objectos da memória, devemos usar:

```
//removidos do parent/displaylist
painel.addChild(combo);
painel.addChild(texto);
//removendo da memória:
combo=null;
texto=null;
```

Ou seja, devem sempre remove-lo da memória, caso contrário vai sobrecarregar a nossa aplicação.

Se por um lado remover objectos com o removeChildAt() pode ser rápido, por outro lado pode ser bastante penoso e complicado pelo simples facto que se tivermos 20 child's no container, poderemos remover o child errado alem de ser complicado saber o numero do respectivo, para isso existe algo que podemos usar para simplificar o processo; A propriedade **.name** dos objectos que identificam o objecto pelo nome na displayList, na função em cima bastaria dar os nomes aos 3 componentes (painel, comboBox e TextInput) :

antes de os adicionar-mos como childs, definimos:

```
painel.name=”meu_painel1”;
combo.name=”meu_combo1”;
texto.name=”meu_texto1”;
```

e para obter o numero do child painel, usariamos:

```
private var objeto:DisplayObject = getChildByName("meu_painel1");
private var num_painel:int = getChildIndex(objeto);
```

e finalmente para o remover:
removeChildAt(num_painel);

o mesmo se passaria com os childs combo e texto:

```
private var objeto_combo:DisplayObject = painel.getChildByName("meu_combo1");
private var num_combo:int = getChildIndex(objeto_combo);

painel.removeChildAt(num_combo);

private var objeto_texto:DisplayObject = painel.getChildByName("meu_texto1");
```

```
private var num_texto:int = getChildIndex(objeto_texto);

painel.removeChildAt(num_texto);
```

Notem que aqui coloquei o **painel. t**anto no getChildByName como no removeChild, visto que os child's combo e texto são filhos do painel.

4.4. Mudado a ordem dos childs na display List

Se por algum motivo quiser-mos trocar a ordem dos childs na display list, às vezes podemos querer faze-lo, por exemplo ao adicionar um botão à aplicação e adicionar-mos depois um painel, o botão ficará coberto pelo painel, assim para colocar-mos o botão à frente do painel bastaria trocar a ordem deles na display list, vejamos:

```
private var btn:Button = new Button;
private var pnl:Painel = new Painel;

btn.x=25;
btn.width=125;
btn.y=50;
btn.name=”but1”;
addChild(btn);

pnl.x=20;
pnl.y=40;
pnl.width=250;
pnl.height=200;
pnl.name=”painel1”
addChild(pnl);
```

Neste caso apenas veriamos o painel na nossa applicação, ou seja, para vermos o botão teriamos ou que adicionar como filhos por outra ordem:

```
addChild(pnl);
addChild(btn);
```

podemos ver as respectivas posições dos nossos componetes assim:

```
trace("->"+getChildAt(0).name);
trace("->"+getChildAt(1).name);
```

e veremos no output:
-> but1
-> paine1

Para os mudarmos na ordem (z), podemos usar as seguintes formas:

```
swapChildren(btn, pnl);
```

que troca a ordem dos componentes no eixo zz identificando pelos componentes, ou

swapChildrenAt(0, 1);

que os troca, identificando pelos seus numeros/index's ou ainda:

btn.setChildIndex(2);

que coloca o nosso botão com o numero/index 2, ou seja, acima do pnl (index=1).

Estas são algumas formas que permitem alterar o index dos componentes na displayList.

5. Programação Orientada a Objectos.

Por muitos a liguagem orientada a objectos é uma maneira simples, rápida e eficaz que se pode traduzir por uma melhor performance no código e uma mais fácil identificação de erros bem como a interpretação por parte de outros programadores. Mas como tudo, também existe outro lado da moeda, que muitos programadores ainda mostram barreira em aceitar esta metodologia de programação e se mostram muito reticentes ao seu uso.
Mas no Action Script 3 não existe outra maneira, ou se programa orientado a objectos ou então é melhor esquecer o AS3.

Daqui para a frente falaremos cada vez mais de este tipo de programação, bem como o seu uso, por isso vamos falar em mais concreto dos beneficios de usar a adequada estruturação do código. Um dos básicos da linguagem orientada a objecto é a compreensão do seu método de funcionamento e estruturação, com especial atenção a:

Classes: Colecção de funções e variáveis relacionadas para facilitar a interacção com determinados objectos. Isto é o 'core' do AS3 e das linguagens OOP, e será mostrado o seu uso em diferentes situações.

Inheritance (Código hereditário) : Uma das grandes vantagens do AS3 (OOP) é a possibilidade de alterar/remover/adicionar propriedades/funções a determinado componente/função/método sem ter que re-inventar a roda, o que nos permite extender uma class/componente sem termos que programar a class do zero, o que além de nos poupar imensas linhas de código, permite também entender melhor o comportamento de determinado componente ou class e melhora-las/adequa-las às nossas necessidades.

Composition (Composição): Como foi falado em cima, esta possibilidade de extender todos os componentes e suas class's, pode ser muito util, mas não é possível em algumas situações e para isso temos este metodo "Composition" que é uma técnica que nos permite criar uma class que possa trabalhar com outra em conjunto, e que nada herda de uma possível class de um componente.

Encapsulation: Geralmente não é uma boa ideia expor todos os aspectos de uma class para outras class's ou aplicação, esta Ecapsulação é um método que permite "ocultar" a maior parte das propriedades e funçoes de uma class permitindo apenas que certos elementos sejam expostos.

Polymorphism: Este método permite que objectos de diferentes class's possam ter o mesmo nome e os mesmos eventos e que se comportem de diferentes formas, o que acaba por permitir que não seja necessário que se criem novos métodos sendo depois mais facil extender essa class...tendo apenas que perceber como o objecto se comporta nas diferentes classes.

Nota: É importante compreender que as linguagens OOP podem não ser fáceis, nem apropriadas para toda a gente ou situação, mas é muito útil principalmente quando usada em projectos de média/grande dimensão com bastantes programadores, já que simplifica em muito o trabalho de equipa. Não é obrigado a aprender OOP para saber programar AS3, mas que ficará limitado a quase 50% das possibilidades do AS3 também é bem verdade.

Uma das iniciais vantagens no processo de programar uma linguagem OOP é escolher uma boa plataforma para o seu desenvolvimento, pricipalmente porque existem variadas maneiras de criar uma interface e várias ferramentas para o fazer. Uma boa prática é que antes de iniciar o seu projecto faça o seu plano de necessidades, objectivos, arquitectura e comportamento de programação. Se fizer isto, é meio caminho andado para perceber que uma linguagem OOP como o AS3 pode satisfazer facilmente as necessidades do projecto, mas não se preocupem em ser demasiado perfecionistas nos vossos primeiros projectos, porque com certeza que encontrarão problemas e cometerão erros, que servirão para optimizar e melhorar a vossa maneira de trabalhar.

5.1. Class's

As class's são a base de qualquer linguagem OOP, que oferecem uma vantagem e melhoramento enorme em relação à programação não orientada a objectos, e que provavelmente já foram usadas por grande parte dos programadores, mesmo não tendo a intenção de as usar, até mesmo nas partes anteriores deste tutorial já foram usadas class's indirectamente e se calhar nem repararam. Ao terem aprendido a trabalhar com eventos (addEventListner, removeEventListner e mesmo o eventDispatch), com a displayList (Display Object, e Display Object Container), e até mesmo com arrays que no seu 'core' todos este métodos/propriedades são class's.

Não fiquem preocupados se estão a pensar que não compreenderam o porque do que falei em cima serem class's porque mais à frente irão compreender, até porque estas class's não estavam "visíveis" quando as utilizamos o que por isso acabamos por não ter percebido que eram class's.

Por exemplo, utilizamos anteriormente propriedades e métodos do Button, como o .width, .x, .y, move() e sem terem percebido, todas estas propriedades pertencem à class Button até aprenderam como criar uma nova instancia dessa class:
var btn:Button = new Button();
que é o fundamental para começar a trabalhar com class's. Então (já) com toda essa experiência em class's qual é a dificildade?? A principal dificuldade explicada mais à frente prende-se como o facto de programar essas mesmas classes, porque como viram é muito fácil usa-las e já sabem bem como o fazer.

Iremos por começar por criar uma "custom class", ou seja, uma class personalizada, vejam:

```
package teste
{
	import mx.controls.Button;

	public class teste extends Button
	{
		public function teste()
		{
		trace("Botão");
		}
	}
}
```

Esta é a estrutura base de uma class hereditária pelo que vamos passar a explicar detalhadamente:
Iniciamos a declaração package indicando o seu nome (teste), não é literalmente obrigatória, mas começamos já assim para explicar bem as coisas, logo depois usamos o import que indica à nossa class que vamos fazer uso de uma outra class (Button), se não colocar-mos esse import o compilador não saberá nem indentificará esta class como uma "extensão" de outra, e simplesmente não funcionará.
Indicamos então a iniciação da nossa class do package, (public class test) e vamos extender a class button (extends Button), que fará com que todas as propriedades, métodos e eventos da class buttom estejam disponiveis para uso na nossa class.
Por final temos a função/método base da nossa class. (public function teste)

Para usarmos esta class, bastaria fazermos no nosso projecto:

```
import teste.teste;
var meubtn:teste = new teste;
```

é familiar não é? Esta declaração é exactamente igual a qualquer outra declaração de objectos, pelo que agora podem perceber o quanto simples é criar uma class. Como resultado desta nossa class, a variável meuBtn será exactamente igual a um Button porque a nossa class teste extende um button e mais nada faz, pelo que se usarem depois do código em cima:

```
addChild(meuBtn)
```

será apresentado um botão no stage, exacatmente igual a um Button normal, ou seja, não notarão qualquer diferença entre:

```
import teste.teste;
var meubtn:teste = new teste;
addChild(meubtn);
```

e

```
var meubtn2:Button = new Button;
addChild(meubtn2);
```

5.1.1. Class Paths (caminhos das class's)

O caminho das class's pode-se entender como sendo o caminho físico onde a class se encontra, no exemplo em cima, ao definir package teste, estamos a definir também o caminho desta class, que ficou organizada no sistema de ficheiros (em directorias) como:

```
->Projecto:
   ->teste (package)
      ->teste (class)
```

Assim facilmente entendemos que poderiamos criar outra class neste package test, fazendo o seguinte:

```
package teste
{
        import mx.containers.Panel;

        public class exemploPanel extends Panel
        {
                public function exemploPanel()
                {
                        trace("exemploPanel");
                }

        }
}
```

E usariamos:

```
import teste.exemploPanel;
var meuPnl:exemploPanel = new exemploPanel;
```

Se usarmos ambas as class's (exemploPanel e test), teríamos que importar ambas:

```
import teste.exemploPanel;
import teste.teste;
```

mas isto pode ser simplificado dizendo ao compilador que queremos ambas as classes disponíveis para uso no nosso projecto, para isso em vez dos 2 imports devemos usar:

```
import teste.*;
```

e desta forma, todas as class's do nosso package teste serão importadas.

Depois de criar esta segunda class no mesmo package, a estrutura de directorias será:

->Projecto:
  ->teste (package)
    ->teste (class)
    ->exemploPanel (class)

Facilmente identificável e separadas para uma melhor organização e compreensão do nosso projecto. O básico das class's está explicado e já é possível pensar em inúmeras utilidades para este tipo de programação, principalmente na sua facilidade de uso.

5.2. Inheritance.

Na class em cima, apenas derivamos/extendemos uma class, mas não fizemos qualquer alteração à sua estrutura base, pelo que não usámos nenhuma propriedade herdada quer do Button ou do Panel. O que vamos passar a aprender é como usar este método e compreender o quanto simples é.

Vejamos o seguinte código:

```
package teste
{
	import flash.events.Event;
	import mx.containers.Panel;
	import mx.events.FlexEvent;

	public class teste extends Panel
	{
		[Embed("flexBtn.png")]
		private var iconImg:Class;

		public function teste()
		{
		this.width=270;
		this.height=270;
		this.x=0;
		this.y=0;

		this.addEventListener(FlexEvent.CREATION_COMPLETE, criado, false, 0 ,true);
		}

		private function criado(evt:FlexEvent):void {
			this.titleIcon=iconImg;
			this.dispatchEvent(new Event("btnsProntos"));
			this.removeEventListener(FlexEvent.CREATION_COMPLETE, criado);
		}
	}
}
```

Neste exemplo, usamos uma propriedade herdada do Pai da class (Panel), a titleIcon, para definirmos de imediato um icon no nosso painel assim que ele é criado, e usamos também um eventDispatcher para avisar quando esse title icon foi terminado, basta usarem:

```
import teste.teste;
import mx.controls.Alert;
private function init():void {
	var myPanel:teste = new teste;
	myPanel.addEventListener("btnsProntos", terminado);
	addChild(myPanel);
}
private function terminado(evt:Event):void {
	Alert.show("Painel e icon completos.")
}
```

e ao chamarem a função init, será adicionado um painel à display list já com o nosso icon no topo do painel.

Até aqui apenas fizemos uso de algumas propriedades herdadas do nosso panel, como o .x .width .titleIcon e assim ficaram a perceber que a nossa class é tratada como se fosse um painel. O que vamos agora aprender é como juntar algumas propriedades e métodos pessoais à nossa Class e devido Package.
Vejamos o seguinte código devidamente comentado:

```
package teste
{
	import flash.events.Event;
	import mx.containers.Panel;
	import mx.controls.Label;
	import mx.events.ChildExistenceChangedEvent;
	import mx.events.FlexEvent;

	public class teste extends Panel
	{
		//colocamos a imagem como embed (incoporada no nosso aplicativo e definida
		//com a variavel imediatamente em baixo (iconImg) disponível na forma de class.
		[Embed("flexBtn.png")]
		private var iconImg:Class;
		/*Esta propriedade estará disponivel como elitura/escrita por exemplo: meuPainel.nomePessoal */
		[Bindable]
		private var _nomePessoal:String;
		//Disponivel como dataCriacao em modo de leitura apenas; meuPainel.dataCriacao
		private var _dataCriacao:String;
		//variavel temporaria usada para saber as horas/minutos/segundos usados em baixo
		private var dados:Date = new Date;

		public function teste()
		{
		//definimos o painel
		this.width=270;
		this.height=270;
		this.x=0;
		this.y=0;
```

```
//adicionamos o eventListner
this.addEventListener(FlexEvent.CREATION_COMPLETE, criado,false,0,true);

}

private function criado(evt:FlexEvent):void {
/*definimos a hora exacta da criação do painel, com recurso ao date, neste caso d variável dados */
        _dataCriacao=dados.getHours()+":"+dados.getMinutes()+":"+dados.getSeconds();
            //adicionamos o icon ao painel
            this.titleIcon=iconImg;
            //verificamos se o nomePessoal foi definido
            if(!_nomePessoal) {
            /*se o nome nao tiver sido dados, damos um nome temporário, no caso será dados como:
            Teste Painel "numero" onde o numero será um valor aleatório entre 0 e 15, este
            numero é conseguido através do Math.floor(Math.ramdom()*15); */

            _nomePessoal="Teste Painel "+Math.floor(Math.random()*15);
            }
            /*vamos usar um label que indica o nomePessoal e a hora da criação do painel, este
            Label desaparecerá assim que forem adicionados filhos ao painel, para isso usamos um
            evenListner
            */
            var lbl:Label = new Label;
            lbl.name="status";
            lbl.x=0;
            lbl.y=0;
            //definimos o texto
            lbl.text=_nomePessoal + " criado em: "+dataCriacao;
            //adicionamos como child ao painel
            this.addChild(lbl);
            //event listner para sabermos quando foi adicionado um filho ao painel para
            //podermos remover o label colocado em cima.
            this.addEventListener(ChildExistenceChangedEvent.CHILD_ADD, remove, false, 0, true);
            //despacha o evento terminado personalizado
            this.dispatchEvent(new Event("btnsProntos"));
            //remove o eventListner terminao, visto que apenas será disparado uma vez.
            this.removeEventListener(FlexEvent.CREATION_COMPLETE, criado);
}

private function remove(evt:ChildExistenceChangedEvent):void {
//função para remover o label com o nome e hora de criação, isto acontece quando é adicionado
//um filho ao painel.
//removemos o child 0 (é o child lbl usado em cima, é o numero zero porque nada mais foi adicionado
ao painel)
            this.removeChildAt(0);
            //removemos o event listner caso contrario se forem adicionados mais que um filho
            //esta função vai remover sempre o ultimo inserido
            this.removeEventListener(ChildExistenceChangedEvent.CHILD_ADD, remove);
}

/*como quero apenas que a propriedade de dataCriacao seja apenas de leitura, fazemos
apenas a função para ler esse valor, com o get, chamado getter */

public function get dataCriacao():String {
            return _dataCriacao;
}
/*como quero que a propriedade _nomePessoal seja quer de leitura, quer de escrita, tenho que
```

```
                dizer à nossa classe que pode disponibilizar ou receber conteúdo de/nessa variável, fazendo isso com
                as funções get e set **/

                public function get nomePessoal():String {
                        return _nomePessoal;
                }
                //Recebe uma string como nome
                public function set nomePessoal(nome:String):void {
                        _nomePessoal=nome;
                }


        }
}
```

No exemplo em cima usei já uma class mais complexa, mas como está devidamente comentada será bem fácil de perceber. Neste exemplo podemos entender o seguinte; É usado um Painel como base, (extends) e as suas propriedades...depois disso é automaticamente adicionado um titleIcon exemplificando atributos herdados do pai (Panel).
O que acrescentei depois foram alguns novos atributos ao nosso Painel Pessoal, como o nomePessoal e o dataCriacao, onde ambos podem ser acedidos para ler os seus valores:
trace(meuPainel.nomePessoal);
trace(meuPainel.dataCriacao);

e o nomePessoal pode ser definido:

meuPainel.nomePessoal="nome pessoal do meu painel";
Esta leitura/escrita destes valores só é possivel devido aos getter's e setter's que estão disponíveis nas nossas class's, onde o get é usado em:

trace(meuPainel.nomePessoal);

e o set:

meuPainel.nomePessoal="nome pessoal do meu painel";

Neste exemplo em cima também vemos o uso de algumas funções matemáticas (floor e ramdom) que podem ser acedidas pela class Math. De seguida adicionamos um label ao painel assim que ele é criado com o nomePessoal e a hora da sua criação, e ao juntar um eventListner esse label será eliminado assim que seja adicionado um child ao painel. Não esquecendo também que removemos os eventListners para não causar conflito.

Para testarem esta class basta fazerem:

```
import mx.controls.Button;
import teste.teste;
import mx.controls.Alert;

private var myPanel:teste = new teste;
```

```
private function init():void {
        myPanel.addEventListener("btnsProntos", terminado);
        //myPanel.nomePessoal="Painel GRUN"; podem ou não definir o nome pessoal do nosso painel.
        addChild(myPanel);
        }

private function terminado(evt:Event):void {
        Alert.show("Painel e icon completos.")
        }

private function addChildMe():void {
        var temp:Button = new Button;
        myPanel.addChild(temp);
        }
```

Aqui se chamarem a função init, o nosso painel é criado no stage, com o nome e hora da criação... se chamarem a função addChildMe será adicionado um botão ao nosso painel e o label que lá existia será removido automaticamente. E pronto, finalmente têm a vossa class criada que é nada mais nada menos um painel personalizado, e sem darem muito por isso já usaram métodos e parametros hereditários, bem como a composição que anteriormente foi explicada..

Continua...