Introduction to ActionScript 3.0 and MXML

obecto

passionate about software & people

Overview of the Compilation Process

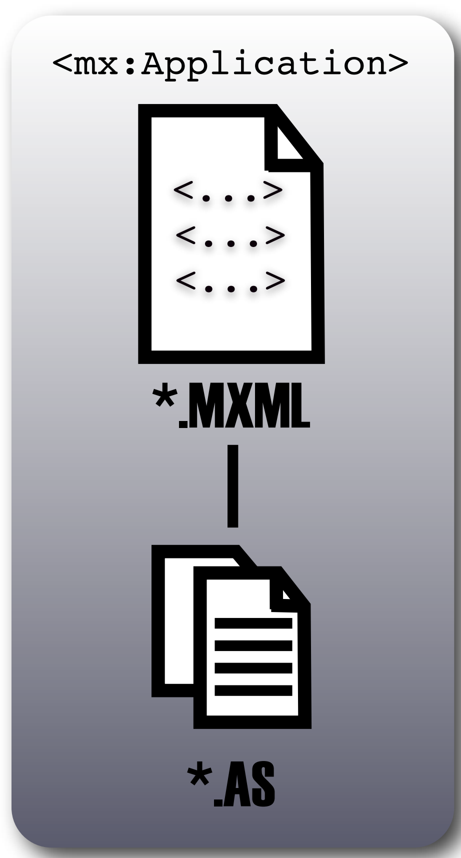
main.mxml

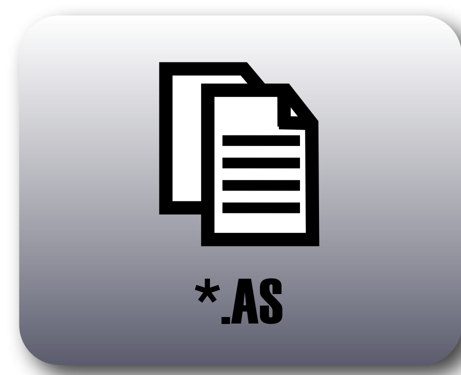
Custom components

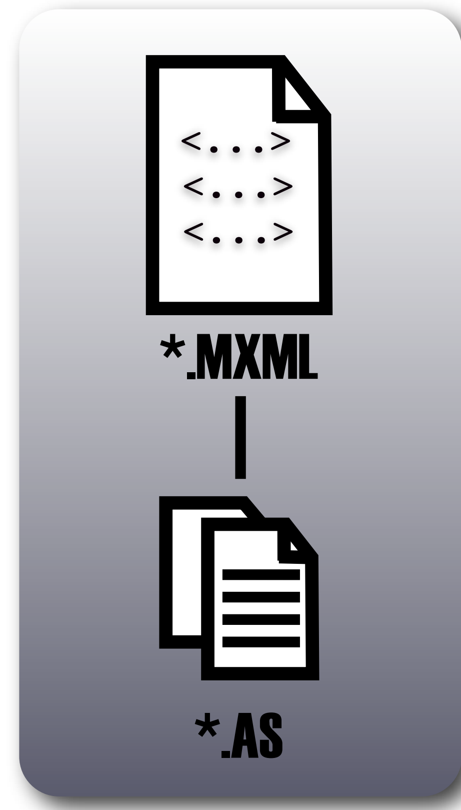
Compiler/Linker

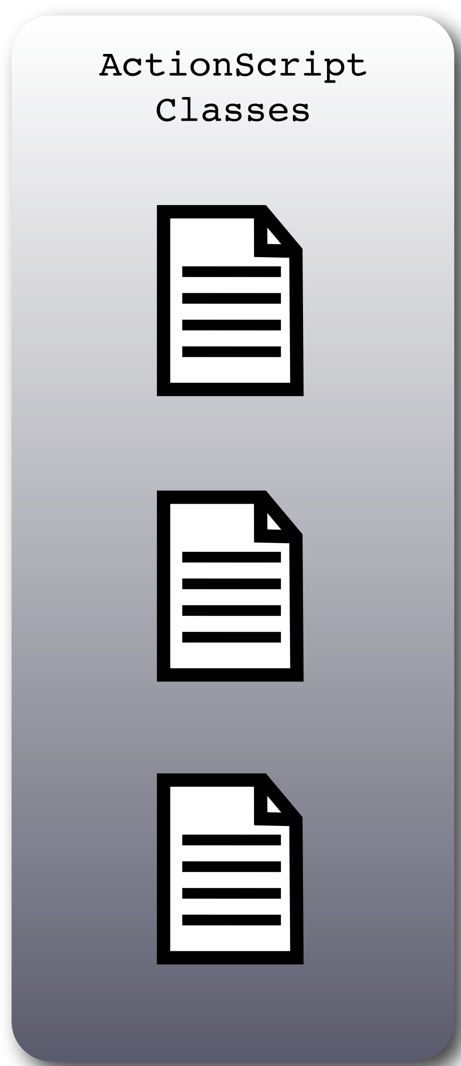
Web server

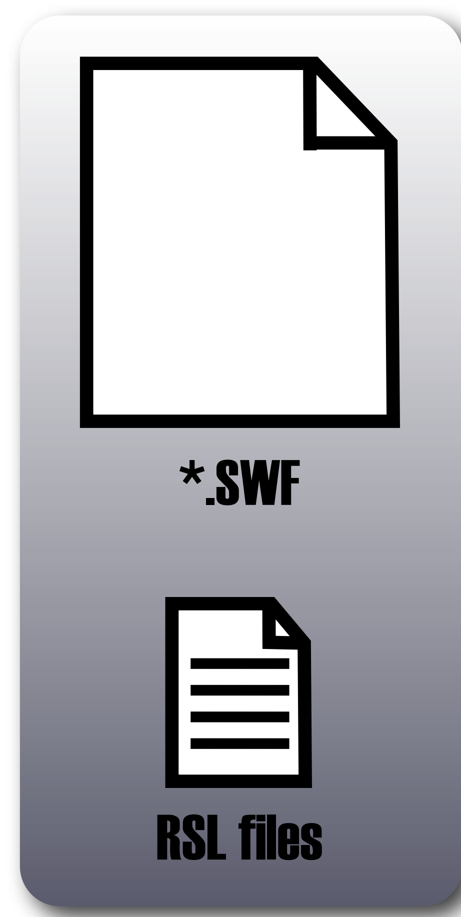
Client

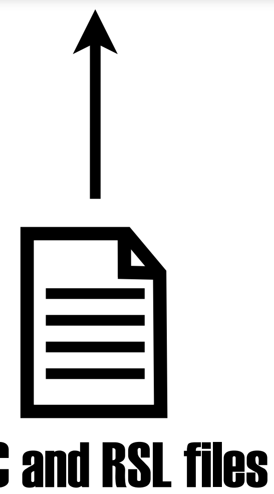












obecto
passionate about software & people

Classes & Interfaces

obecto
passionate about software & people

```
[dynamic] [public | internal] [final] class className
  [ extends superClass ]
  [ implements interfaceName[, interfaceName... ] ] {
  // class definition here
}
```

- **public** - the class is available to every caller
- **dynamic** - instances of the class may have dynamic properties added at runtime
- **final** - the class can not be extended
- **internal** - the class is available to every caller within the same package

MXML is also a Class

CustomCanvas.as

```
package {
    public class CustomCanvas
        extends Canvas
        implements IListItemRenderer,
            IFocusManager
    {
        public function CustomCanvas()
        {
            // constructor
        }
    }
}
```

CustomCanvas.mxml

```
<mx:Canvas
    implements="IListItemRenderer, IFocusManager"
    xmlns:mx="http://www.adobe.com/2006/mxml">

    <!-- mxml definitions -->

</mx:Canvas>
```

obecto
passionate about software & people

```
interface InterfaceName [extends InterfaceName]
```

- **you can define an interface only in ActionScript**
- **interfaces contain *only* declarations of methods**
- **method definitions *can not* have access modifiers as *public* or *private***
- **method implementations must be marked as *public***
- **you can't have properties declarations, but you can declare *getter* and *setter* methods**

```
package packageName {
    class SomeClassName {
    }
}
```

- **indicates that a class is part of a package**
- **packages allow you to organize the code into discrete groups that can be imported**

```
import packageName.ClassName;
import packageName.*;
```

- **makes externally defined classes and packages available to your code**

```
include "[path]filename.as"
```

- **includes the contents of the file, as if the file commands are part of calling script**
- **the *include* directive is invoked at compile time**

XML namespaces as imports

CustomComponent.as

```
package com.obecto.view {
    public class CustomComponent
        extends UIComponent
    {
        // ...
    }
}
```

CustomService.as

```
package com.obecto.service {
    public class CustomService
        extends AbstractService
    {
        // ...
    }
}
```

CustomCanvas.mxml

```
<mx:Canvas
    implements=”IListItemRenderer, IFocusManager”
    xmlns:mx=”http://www.adobe.com/2006/mxml”

    xmlns:view=”com.obecto.view.*”
    xmlns:service=”com.obecto.service.*”>

    <view:CustomComponent/>
    <service:CustomService/>

</mx:Canvas>
```

Functions and Methods

```
function functionName([parameter0,...parameterN]) : returnType{
  // statement(s)
}

var functionRef:Function = function ([parameter0,...parameterN]) : returnType{
  // statement(s)
}
```

- **you can create an *anonymous* function and return reference to it**
- **you can have nested functions**

```
[virtual | override | final] [public | protected | private] [static] function
```

- **override - the method replaces an inherited method (can’t use it on static and final methods)**
- **final - the method can not be overridden**

```
function name(param0 : Number, param1 : String = “DefaultValue”) : returnType
```

- **required parameters are not permitted after optional parameters**

```
function passAnything(...argumentsArray) : void
{
    trace("arguments count: " + argumentsArray.length);
    trace("your arguments are: " + argumentsArray);
}
```

- functions can be defined to have variable parameters count
- empty parameters list means the method can't be invoked with parameters

```
var funRef : Function = function(a : int, b: int) : int
{
    return a + b;
}

funRef(2, 3); // () - function call operator

funRef.call(otherThisObject, 2, 3);

funRef.apply(otherThisObject, [2, 3]);
```

- **you can use a reference to a function as a regular function**
- **to change the *this* reference to refer to other object use *call* or *apply***
- ***apply* is same as *call*, but the arguments are passed in an Array instead of a comma-separated list**
- **why is this useful? is it useful at all?**

obecto
passionate about software & people

Variables & Constants

```
var index : uint = 0;

private var isReady : Boolean = false;

protected var dictionary : Dictionary = new Dictionary();

public static const MINUTES_IN_HOUR : Number = 60;

const weekendDays : Array = [“Saturday”, “Sunday”];
```

Getters

```
function get propertyName():returnType
{
    // statements ending with return
}

trace(propertyName);
```

Setters

```
function set propertyName(value:*):void
{
    // statements without return
}

propertyName = newValue;
```

- combine getters and setters to create [read-write], [read-only] or [write-only] properties
- getters and setters can be overridden in subclasses

obecto
passionate about software & people

Scope caveat

```
public function scopeSamples() : void
{
   var methodScope : String = "Method scope";
   if (true)
   {
      trace(methodScope);

      var conditionalScope : String = "Conditional scope";
   }
   trace(conditionalScope);

   for (var i : int = 0; i < 5; i++)
   {
      var loopScope : String = "Loop scope"; // duplicate variable definition
   }
   trace(loopScope);
   trace(i);

   var loopScope : String = "Method scope";
   trace(loopScope);
}
```

MXML in Further Detail

obecto
passionate about software & people

MXML Tags

CustomCanvas.mxml

```
<mx:Canvas
    xmlns:mx=”http://www.adobe.com/2006/mxml”
    xmlns:view=”com.obecto.view”
    xmlns:service=”com.obecto.service”>

    <view:CustomComponent id=”viewComponent”
        componentName=”TEST!@#”/>

</mx:Canvas>
```

CustomComponent.as

```
package com.obecto.view {
    public class CustomComponent
        extends UIComponent
    {
        public var componentName : String;
    }
}
```

- MXML tags are actually an instantiation of an object
- id is the name of the reference (same as variable names in AS)
- instantiated components must have constructors with only optional parameters or no parameters at all
- you can access and set the component properties as xml-attributes of the MXML tag

MXML Special Tags

```
<mx:Script>
    <![CDATA[

        // ActionScript goes here!

    ]]>
</mx:Script>
```

```
<mx:Metadata>
    [Event(name=”endOfWorld”)]
</mx:Metadata>
```

```
<mx:Binding source=”srcProperty”
    destination=”destProperty”/>
```

```
<mx:Style>
    .mainText {
        font-family: Georgia;
        font-size: 24;
    }
</mx:Style>
```

or

```
<mx:Style source=”path/file.css”/>
```

other special tags are mx:XML, mx:XMLList, mx:Model and mx:Component

Metadata Tags (a.k.a. Annotations)

- provide information to the compiler how your components are used
- you can annotate the whole class definition if the aspect affects the whole component
- you can annotate a single property of the class
- you can annotate with your custom metadata, but you need to compile with the *keep-as3-metadata* option
- you can use the *Reflection API* to get to the metadata

obecto
passionate about software & people

Metadata Tags

- [ArrayElementType]
- [Bindable]
- [DefaultProperty]
- [Embed]
- [Event]
- [Inspectable]
- [InstanceType]
- [RemoteClass]
- [Style]
- [Transient]

```
[ArrayElementType("com.obecto.data.Entity")]
public var items : Array;
```

```
[Bindable]
public var data : Object;
```

```
package com.obecto.view {
    [DefaultProperty("items")]
    public class CustomComponent
        extends UIComponent
    {
        public var items : Array;
    }
}
```

```
[Embed(source="assets/animation.swf")]
public var embededMovie : Class;
```

```
[Event(name="mouseOver", type="flash.events.MouseEvent")]
[Event(name="mouseOut", type="flash.events.MouseEvent")]
public class CustomComponent {}
```

Exception Handling

obecto
passionate about software & people

```
throw new Error("error message");
```

```
try {
    // try block
} finally {
    // finally block
}

try {
    // try block
} catch(error[:ErrorType1]) {
    // catch block
} [catch(error[:ErrorTypeN]) {
    // catch block
}] [finally {
    // finally block
}]
```

Exception handling is exactly the same in Java (no significant difference that worths noting)

Iterators

```
var array:Array = [1, 2 ,3];
for (var i:int = 0; i < array.length; i++)
{
    trace(array[i]);
}
```

The classic for loop

```
var array:Array = [1, 2 ,3];
for each (var i:int in array)
{
    trace(i);
}
```

for each

```
var object:Object = {one:”ONE”, two: “TWO”};
for (var key:* in object)
{
    trace(object[key]);
}
```

Iterating through the properties of an object

obecto

passionate about software & people

Regular Expressions

```
var someName : String = "Jackie Smith";

if (someName.search(/jack/i) != -1)
{
   trace("Jackie's here!");
}
```

Regular expressions are part of the syntax of ActionScript

```
var toungeTwister : String =
    "Peter Piper Picked a peck of pickled peppers";

var pickRegExp : RegExp = /pick|peck/g;

var replaced : String =
    toungeTwister.replace(pickRegExp, "Match");

trace(replaced);
```

Every regular expression is an instance of the RegExp class

E4X (ECMAScript for XML)

obecto
passionate about software & people

```
var sales : XML = <sales vendor="John">
    <item type="peas" price="4" quantity="6"/>
    <item type="carrot" price="3" quantity="10"/>
    <item type="chips" price="5" quantity="3"/>
  </sales>;

trace(sales.item.(@type == "carrot").@quantity);
trace(sales.@vendor);

for each (var item : XML in sales..item ) {
  trace(item.@price);
}
```

- **E4X is a programming language extension that adds native XML support**
- **web services can be returned in E4X format for easy parsing**

obecto

passionate about software & people

Data Binding

- automatic data synchronization between a *source*-property and a *destination*-property
- extension of the *Observer* pattern over the event handling *Observer* implementation
- data binding can be bi-directional

[Bindable]

```
[Bindable]
public var isCompleted : Boolean = false;

[Bindable(event=”itemsChanged”)]
public var itemsCount : Number = 0;
```

- registers a property as bindable (means the registered property can be a *source*-property)
- every change to the value of the property will be automatically copied to the *destination*-property
- the Flex compiler will generate corresponding *getters* and *setters* at compile time
- Flex will automatically turn your class into an *EventDispatcher*
- if we omit the *event*-property in the metadata Flex automatically creates a *propertyChanged*-event

obecto
passionate about software & people

[Bindable]

```
[Bindable(event=”itemsChanged”)]
public var itemsCount : Number = 0;

private var _items : Array;
[Bindable]
public function set items(value : Array) : void
{
    if (value &&
        value.length != _items.length)
    {
        itemsCount = value.length;
        //itemsCount -= fakeElements;
        dispatchEvent(
            new Event(“itemsChanged”));
    }
    _items = value;
}
public function get items() : Array
{
    return _items;
}
```

- **the synchronization of the *itemsCount*-property will be only triggered when the registered *event* is dispatched**
- **you can also register a *getter-setter*-pair as *[Bindable]***

obecto
passionate about software & people

[Bindable]

```
[Bindable]
public class EmployeeInfo
{
    public var firstName : String;
    public var lastName : String;
    public var position : Position;
}
```

a whole class can be registered as *[Bindable]*, so all of its properties will be automatically *[Bindable]*

Curly-Brace Syntax (Binding)

```
<mx:TextInput id=”firstNameInput”/>
<mx:TextInput id=”lastNameInput”/>

<data:CompanyModel id=”model”/>

<data:EmployeeInfo firstName=”{firstNameInput.text}”
    position=”{model.EmployeeOfTheMonth.position}”>
    <lastName>{lastNameInput.text}</lastName/>
</data:EmployeeInfo>
```

- when you register a property as source of data binding, Flex monitors not only the property, but also the *chain of properties* leading up to it

```
<mx:String>{“Concatenating”+bindableVar}</mx:String>
```

- you can have whole expressions inside the binding

```
<mx:Number>{Math.max(width, height)}</mx:Number>
```

- you can call functions will bindable arguments

<mx:Binding>

```
<mx:TextInput id=”firstNameInput”/>
<mx:TextInput id=”lastNameInput”/>

<data:CompanyModel id=”model”/>

<data:EmployeeInfo id=”info”/>

<mx:Binding source=”firstNameInput.text”
    destination=”info.firstName”/>
<mx:Binding source=”lastNameInput.text”
    destination=”info.lastName”/>
<mx:Binding
    source=”model.employeeOfTheMonth.Position”
    destination=”info.position”/>
```

Remember the *Binding* special tag?

obecto
passionate about software & people

BindingUtils

```
BindingUtils.bindProperty(this, “destination”,
    hostComponent, “source”);
```

```
BindingUtils.bindProperty(this, “destination”,
    hostComponent, [“textInput”, “text”]);
```

```
public function update(value : String) : void
{
    textArea.text = value.toUpper();
}

BindingUtils.bindSetter(update, textInput, “text”);
```

- you can use the *BindingUtils* to programatically create bindings
- these methods return a *ChangeWatcher*, which is an utility class that you can use with bindable properties
- to destroy a binding you must call the *unwatch()*-method of the corresponding *ChangeWatcher*

obecto
passionate about software & people

ActionScript

- for non-visual components
- for modifying behavior
- for extending *UIComponent* and other programatic extensions

MXML

- custom component that only add simple behavior
- utilizing the declarative power for composite components
- using MXMLs as *IoC* containers

Summary

- overview of the Flex compilation process
- basic language concepts
- MXML tags and special tags
- metadata tags
- exception handling
- iterators
- regular expressions
- E4X
- data binding
- when to use ActionScript and when to use MXML