CS 175
Action Script

Classes 3

Overriding Functions

- To override a method means to redefine the behavior of an ***inherited*** method.

- To override an instance method, the method definition in the subclass must use the ***override*** keyword and must match the superclass version of the method in the following ways:

  - ➢ The override method must have the same level of access control as the base class method. Meaning if the base class method is public the subclass overridden function needs to be public also (same goes for internal or protected functions).

  - ➢ The override method must have the same number of parameters, same data type annotations and same return type as the base class method.

It will become clearer once we do an example in the coming slides

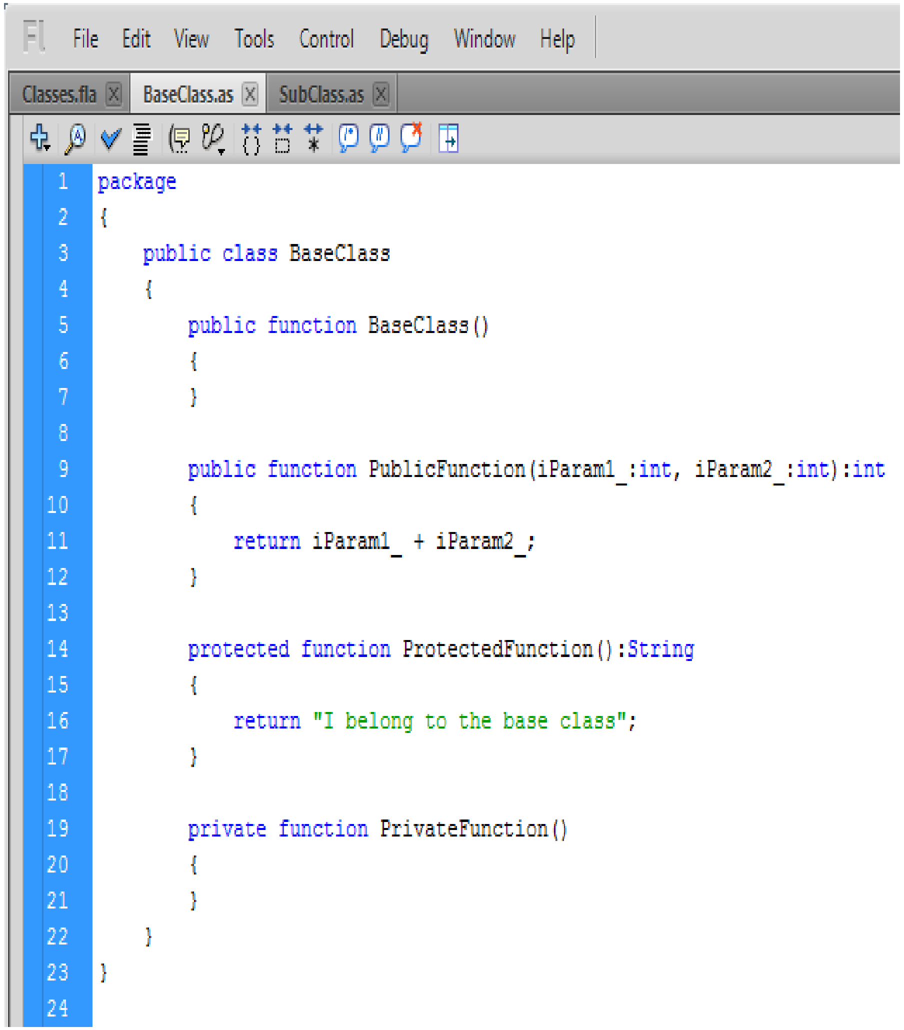
Overriding Functions

- Any instance that isn’t inheritable (***static*** or ***private***) can not be overridden by the subclass.

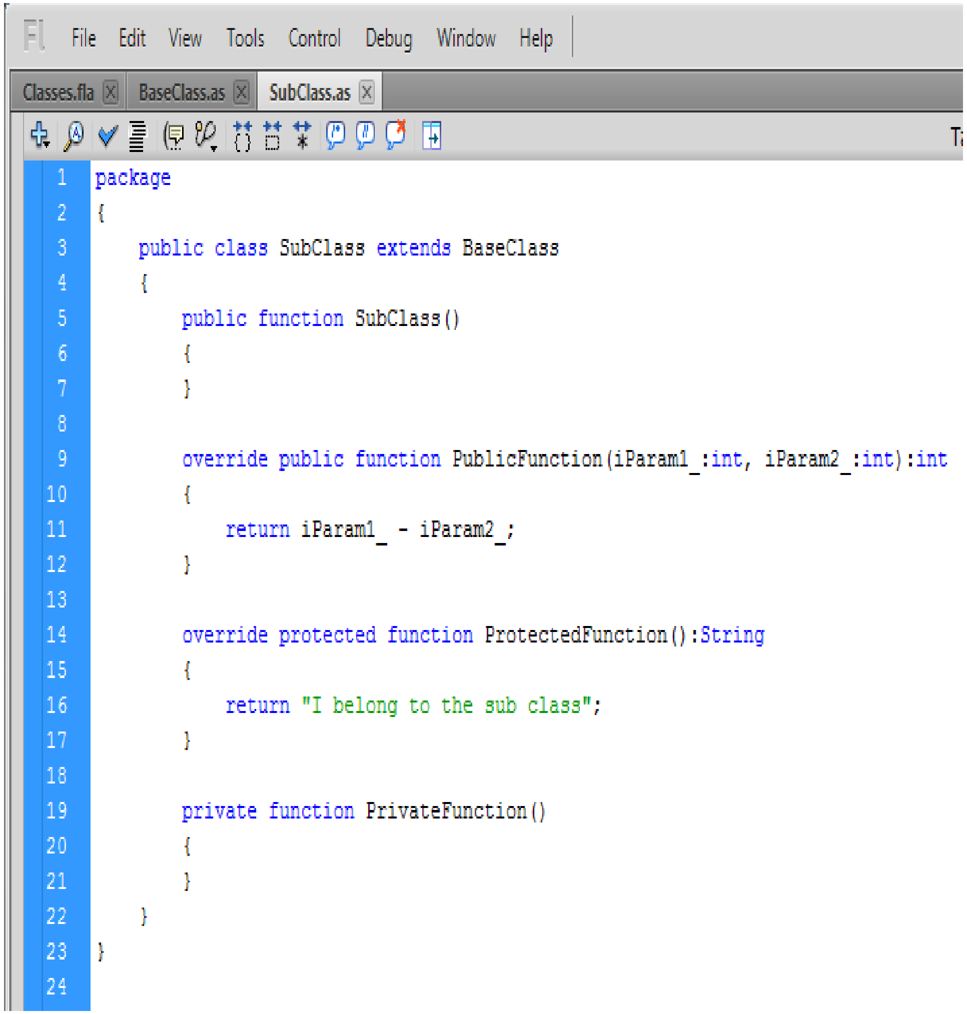
- But of course the subclass can define an identically named method in the subclass, because the base class method will not be visible to the subclass.

Overriding Functions

Example:



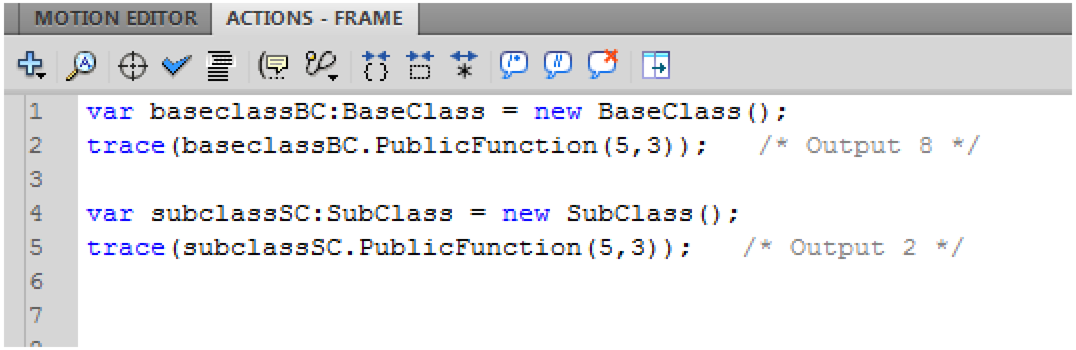
```
package
{
    public class BaseClass
    {
        public function BaseClass()
        {
        }

        public function PublicFunction(iParam1_:int, iParam2_:int):int
        {
            return iParam1_ + iParam2_;
        }

        protected function ProtectedFunction():String
        {
            return "I belong to the base class";
        }

        private function PrivateFunction()
        {
        }
    }
}
```



```
package
{
    public class SubClass extends BaseClass
    {
        public function SubClass()
        {
        }

        override public function PublicFunction(iParam1_:int, iParam2_:int):int
        {
            return iParam1_ - iParam2_;
        }

        override protected function ProtectedFunction():String
        {
            return "I belong to the sub class";
        }

        private function PrivateFunction()
        {
        }
    }
}
```

Overriding Functions

Example (cont’d):



```
var baseclassBC:BaseClass = new BaseClass();
trace(baseclassBC.PublicFunction(5,3));   /* Output 8 */

var subclassSC:SubClass = new SubClass();
trace(subclassSC.PublicFunction(5,3));   /* Output 2 */


```

- As you can see, although the subclass inherited the “PublicFunction” method, it was able to override it and give it a new behavior.

- Again, we can do that with all inherited functions but not with private or static ones.

- With a private function, we can simply redefine a totally new function with the same name in the subclass since it doesn’t already have it.

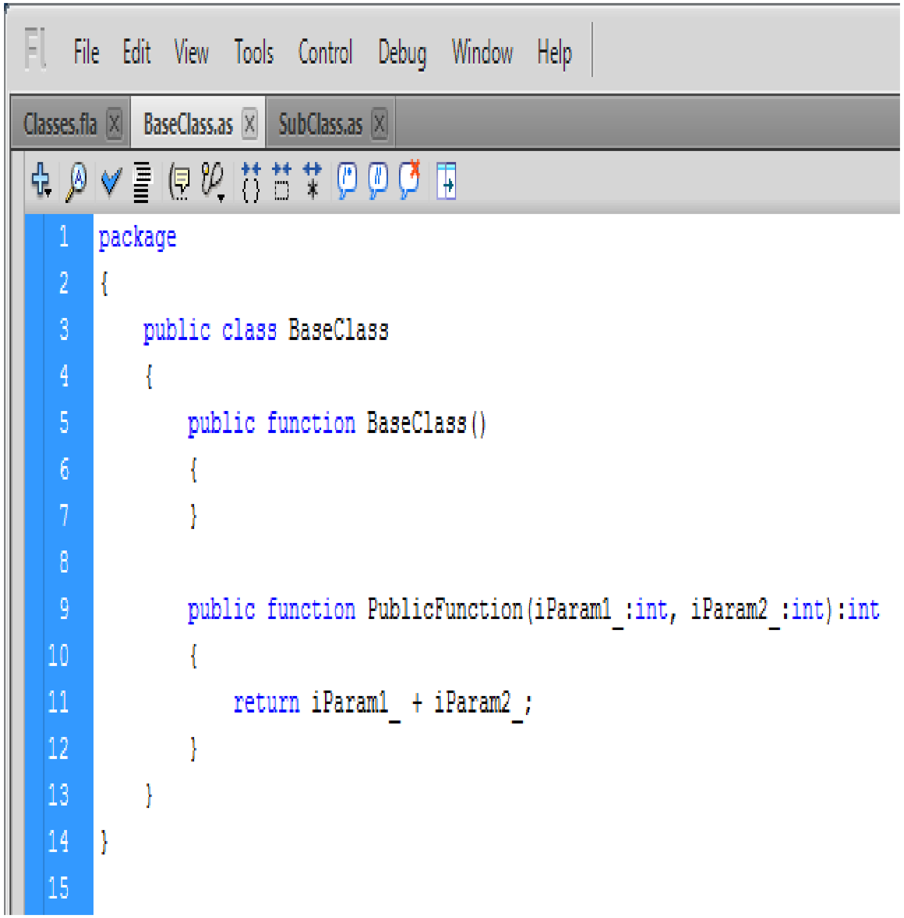
Overriding Functions

Using the *super* keyword in the overridden function:

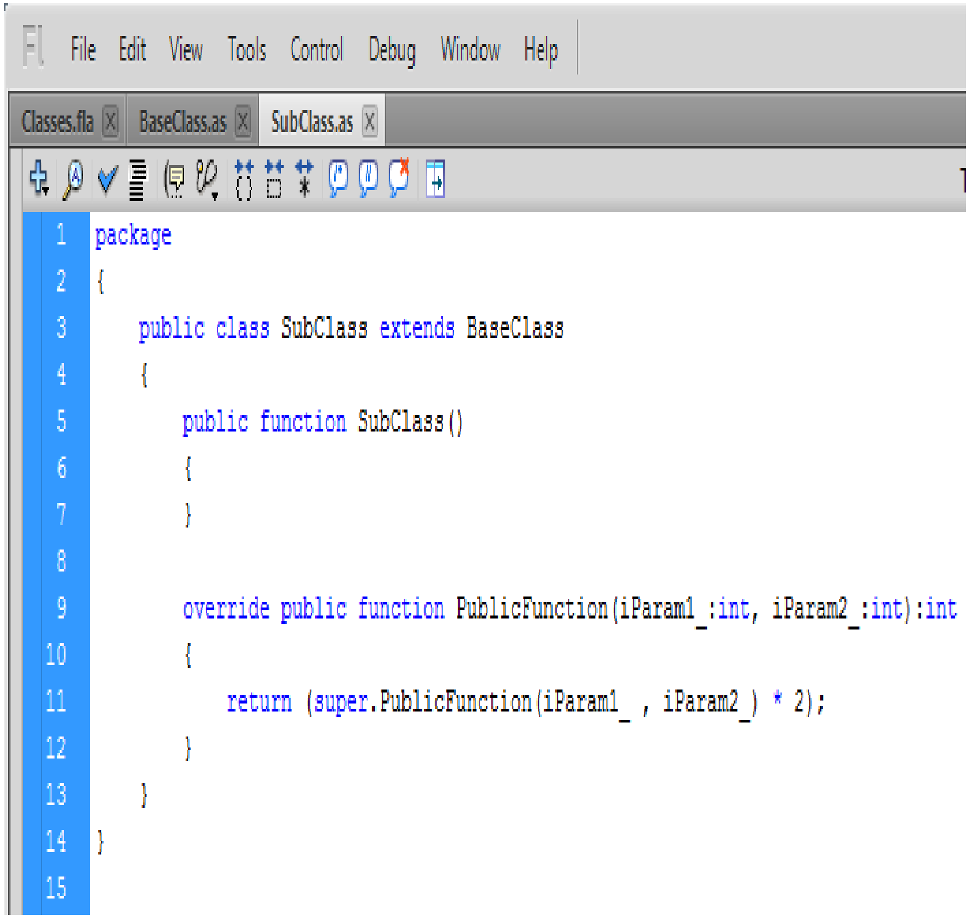
- When overriding a method, programmers often want to add to the behavior of the superclass method they are overriding instead of completely replacing the behavior.
- So basically this requires a mechanism that allows a method in a subclass to call the superclass version of itself.
- The ***super*** statement provides such a mechanism. Using the ***super*** statement you can access the methods defined in the base class including the method you are overloading.

Overriding Functions

Example:



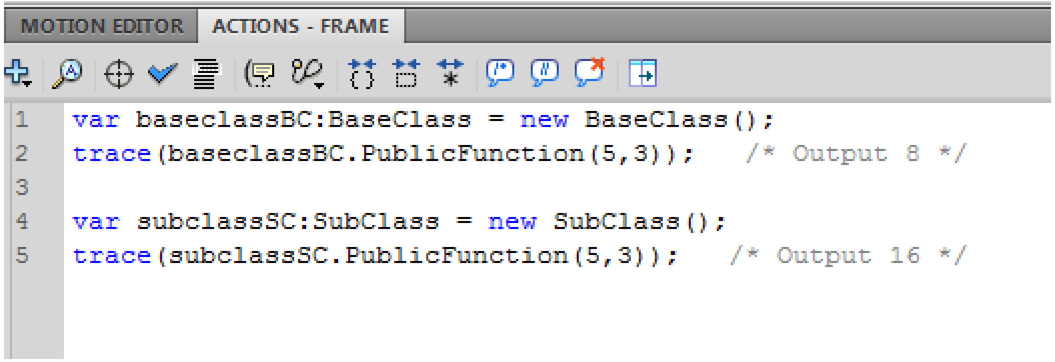
```
package
{
    public class BaseClass
    {
        public function BaseClass()
        {
        }

        public function PublicFunction(iParam1_:int, iParam2_:int):int
        {
            return iParam1_ + iParam2_;
        }
    }
}
```



```
package
{
    public class SubClass extends BaseClass
    {
        public function SubClass()
        {
        }

        override public function PublicFunction(iParam1_:int, iParam2_:int):int
        {
            return (super.PublicFunction(iParam1_ , iParam2_) * 2);
        }
    }
}
```

Overriding Functions

Example (cont'd):



```
var baseclassBC:BaseClass = new BaseClass();
trace(baseclassBC.PublicFunction(5,3));    /* Output 8 */

var subclassSC:SubClass = new SubClass();
trace(subclassSC.PublicFunction(5,3));    /* Output 16 */
```

As you can see, by using the ***super*** statement we were able to use the code in the base class and add more to it in the subclass although we are overriding the function.

Polymorphism

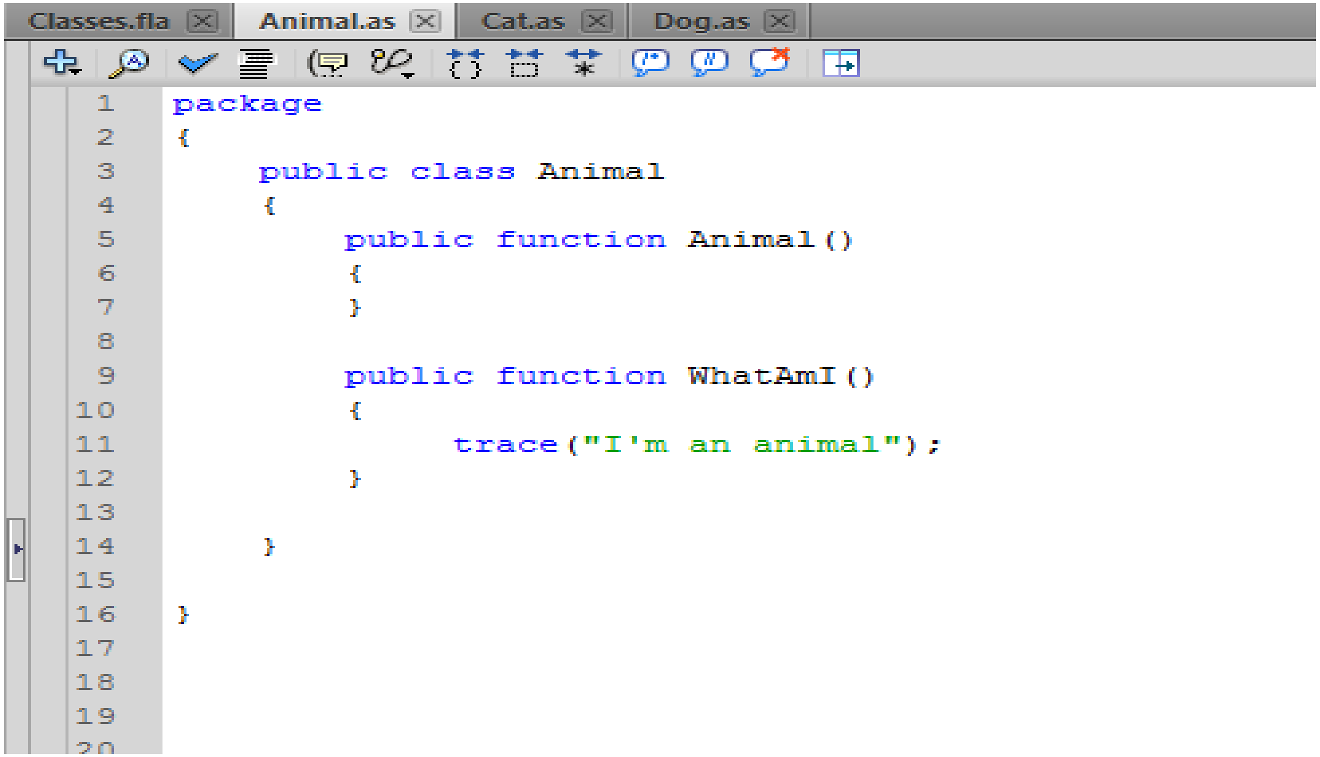
- ***Polymorphism*** is the ability to create an object that has more than one form.

- The purpose of polymorphism is to implement a style of programming in which objects of various types define a common interface of operations for users.

So far it seems hard to understand. Let’s do an example and clarify things.

Polymorphism

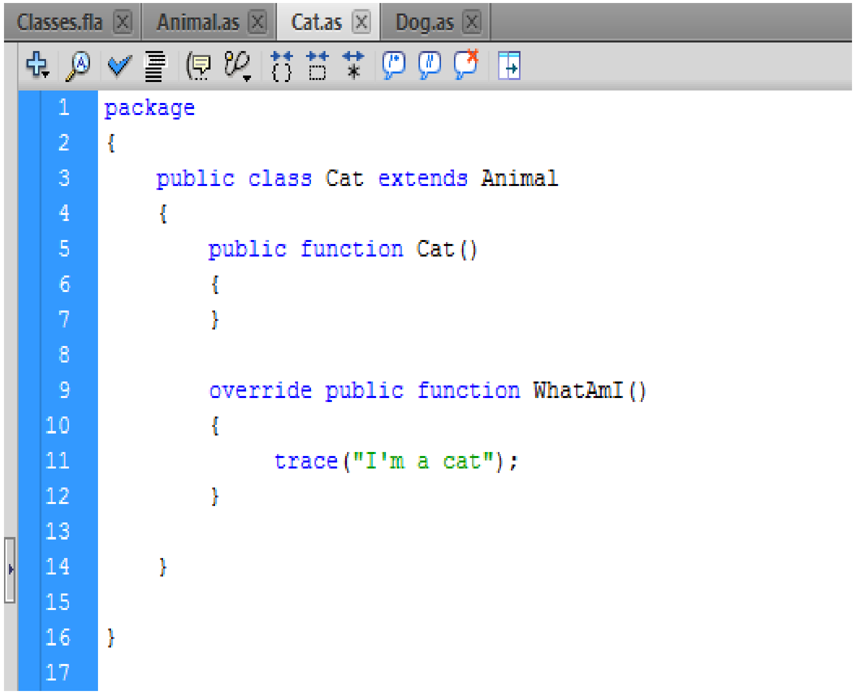
The following will be our base class (superclass):



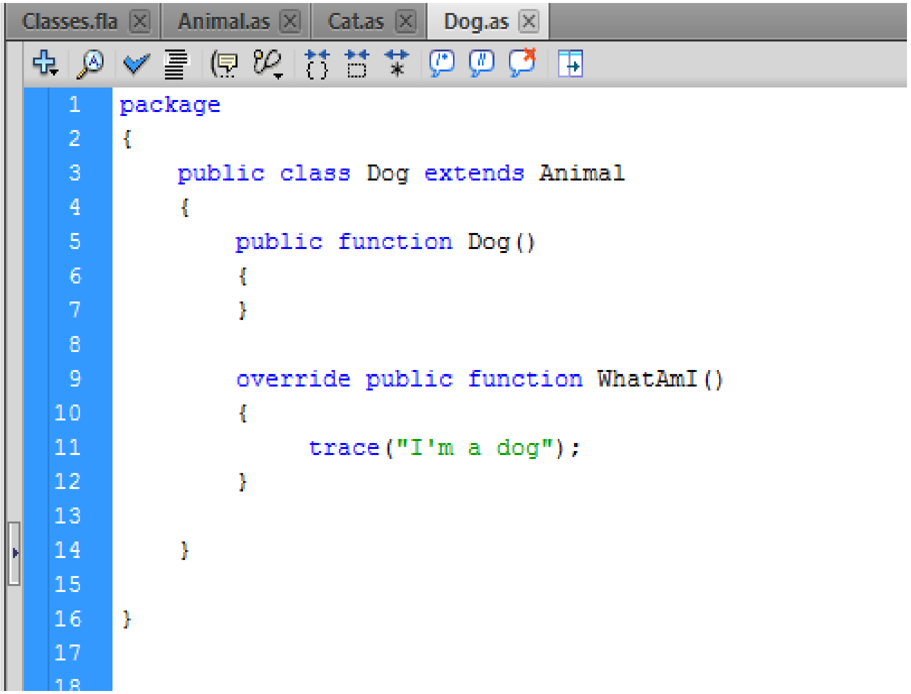
```
package
{
	public class Animal
	{
		public function Animal()
		{
		}

		public function WhatAmI()
		{
			trace("I'm an animal");
		}

	}

}
```

Polymorphism

Derived from the Animal class are the following two classes:



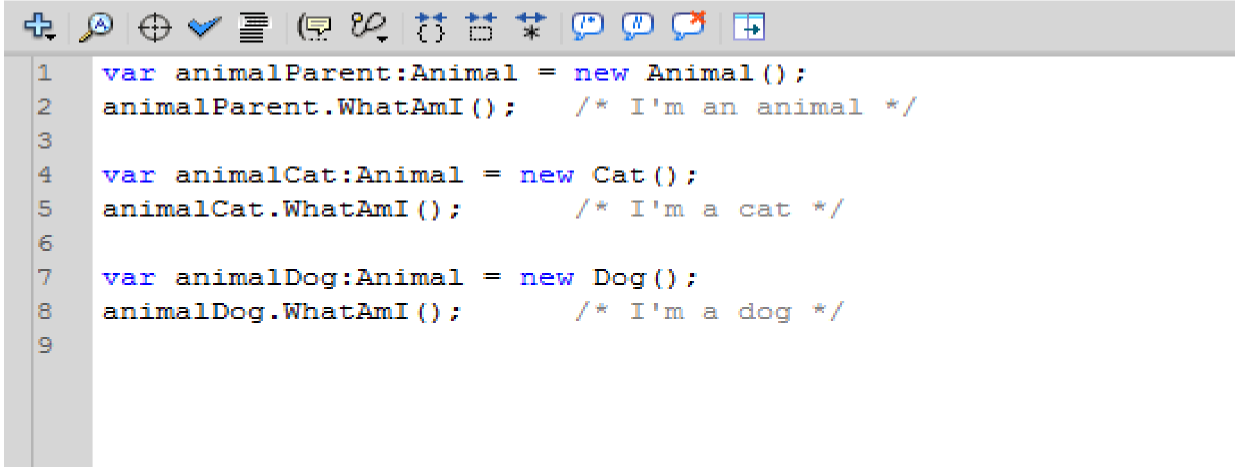
```
package
{
    public class Cat extends Animal
    {
        public function Cat()
        {
        }

        override public function WhatAmI()
        {
            trace("I'm a cat");
        }

    }

}
```



```
package
{
    public class Dog extends Animal
    {
        public function Dog()
        {
        }

        override public function WhatAmI()
        {
            trace("I'm a dog");
        }


    }

}
```

As you can see, both derived classes are overriding the **“WhatAmI”** function.

Polymorphism



```
var animalParent:Animal = new Animal();
animalParent.WhatAmI();    /* I'm an animal */

var animalCat:Animal = new Cat();
animalCat.WhatAmI();        /* I'm a cat */

var animalDog:Animal = new Dog();
animalDog.WhatAmI();        /* I'm a dog */
```

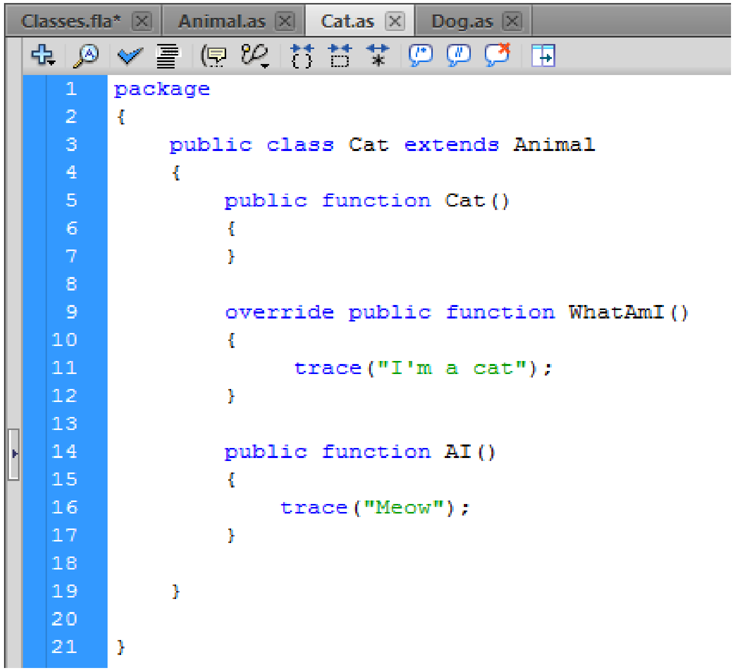
- Using the base class type, we were able to create 3 variables each having his own behavior.

- When we call the overridden function **"WhatAmI"** the variable will know which one to call depending on which constructor we used when creating that variable.

Polymorphism

Important Notes:

- Only common properties between the base class and the subclass can be accessed.



```
package
{
    public class Cat extends Animal
    {
        public function Cat()
        {
        }

        override public function WhatAmI()
        {
             trace("I'm a cat");
        }

        public function AI()
        {
            trace("Meow");
        }

    }

}
```

```
var animalCat:Animal = new Cat();
animalCat.WhatAmI();      /* I'm a cat */
animalCat.AI();   /* error undefined method */
```

- We can't create variables of type the subclass and use the base class constructor.

```
var  catC1:Cat = new Animal();   /* This won't work !!! */
```

Class attributes

- ActionScript 3.0 allows you to modify class definitions using one of the following four attributes:

Attribute	Definition
public	Visible to references everywhere.
internal (default)	Visible to references inside the current package.
final	Must not be extended by another class.
dynamic	Allow properties to be added to instances at run time.

- The ***final*** and ***dynamic*** attributes are used with the ***public*** and ***internal*** ones
  Eg: final public class A, dynamic internal class B

- Not specifying an attribute will be taken as if you specified ***internal.***

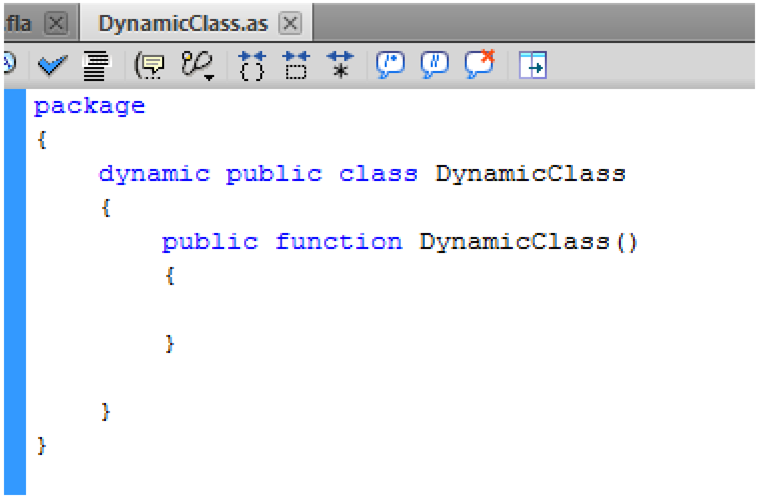
Eg: class A → internal class A
final class B → final internal class B

Dynamic Class

- A class that is not ***dynamic*** is by default a sealed class. You can't add properties or methods to a sealed class at run time.

- A ***dynamic*** class defines an object that can be altered at run time by adding or changing properties and methods.

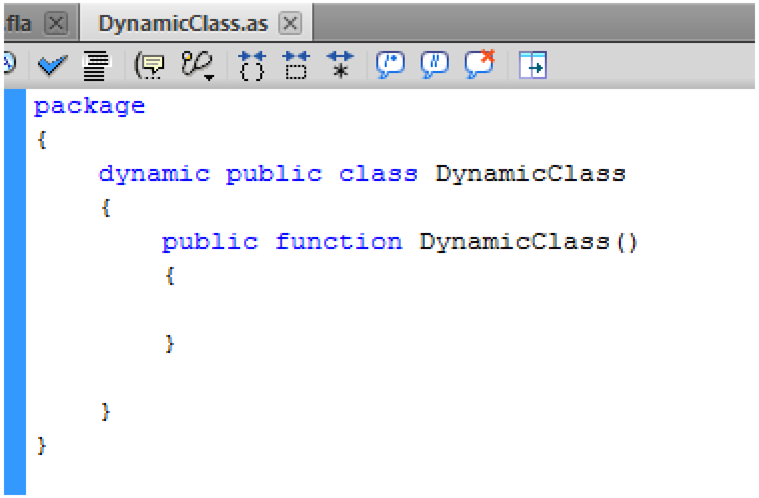
- You create dynamic classes by using the ***dynamic*** attribute when you declare a class.



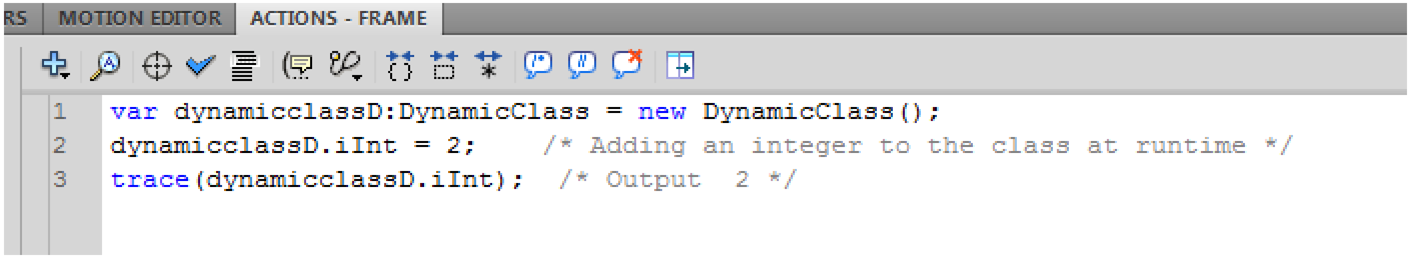
```
package
{
	dynamic public class DynamicClass
	{
		public function DynamicClass()
		{

		}

	}
}
```

Dynamic Class

- You can add properties or methods to a dynamic class outside the class definition



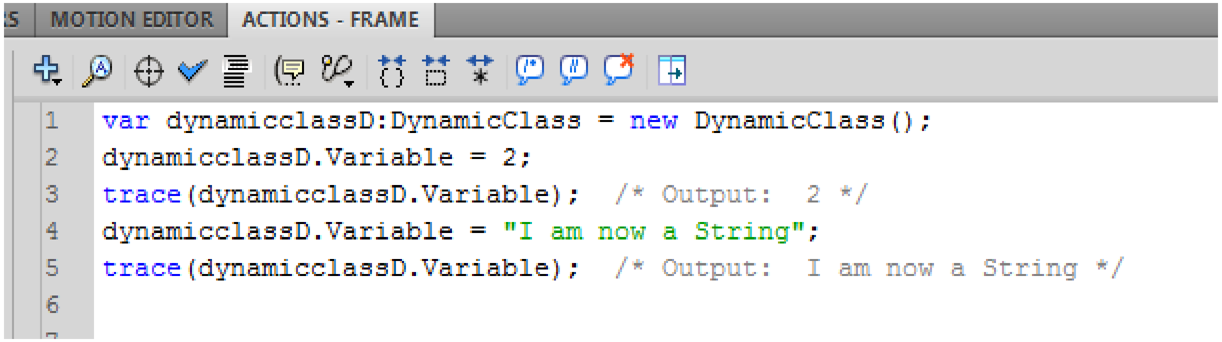
```
package
{
    dynamic public class DynamicClass
    {
        public function DynamicClass()
        {


        }


    }
}
```



```
var dynamicclassD:DynamicClass = new DynamicClass();
dynamicclassD.iInt = 2;     /* Adding an integer to the class at runtime */
trace(dynamicclassD.iInt);  /* Output  2 */
```

Dynamic Class

- You can’t add a type annotation to a property that you add in this manner.

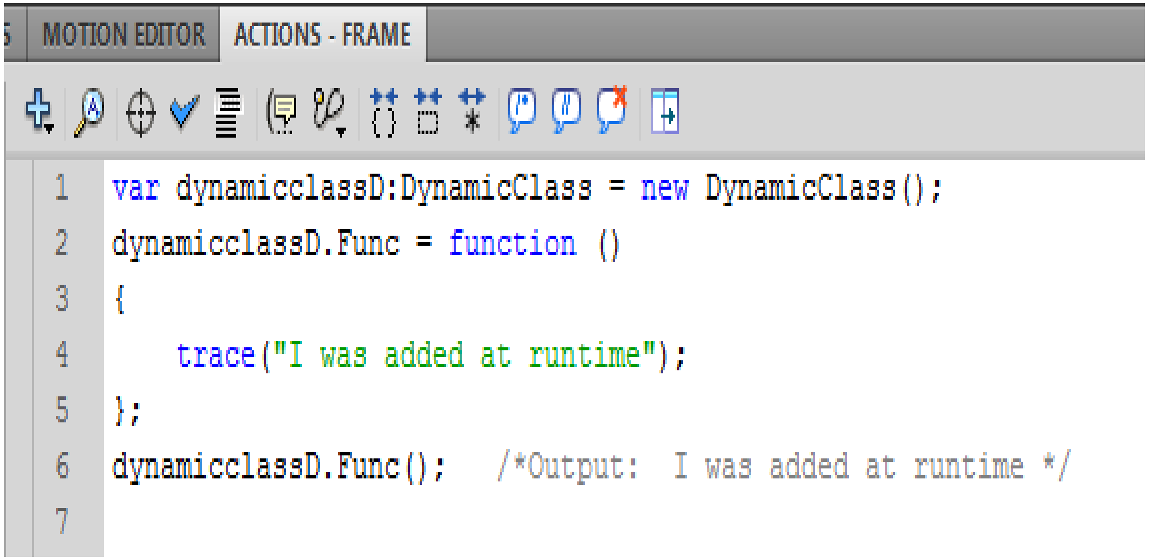


```
var dynamicclassD:DynamicClass = new DynamicClass();
dynamicclassD.Variable = 2;
trace(dynamicclassD.Variable);  /* Output:  2 */
dynamicclassD.Variable = "I am now a String";
trace(dynamicclassD.Variable);  /* Output:  I am now a String */
```

Variable added at run-time are of type **(*)** (No type). So we can change their content to anything at run-time.

Dynamic Class

- You can also add a method to the dynamic class instance by defining a function and attaching the function to a property.

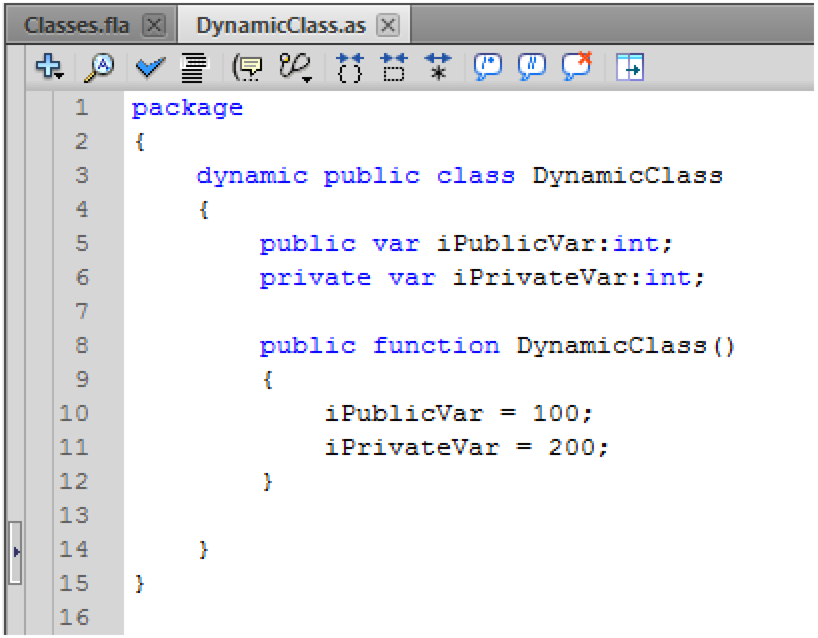


```
var dynamicclassD:DynamicClass = new DynamicClass();
dynamicclassD.Func = function ()
{
    trace("I was added at runtime");
};
dynamicclassD.Func();   /*Output:  I was added at runtime */
```

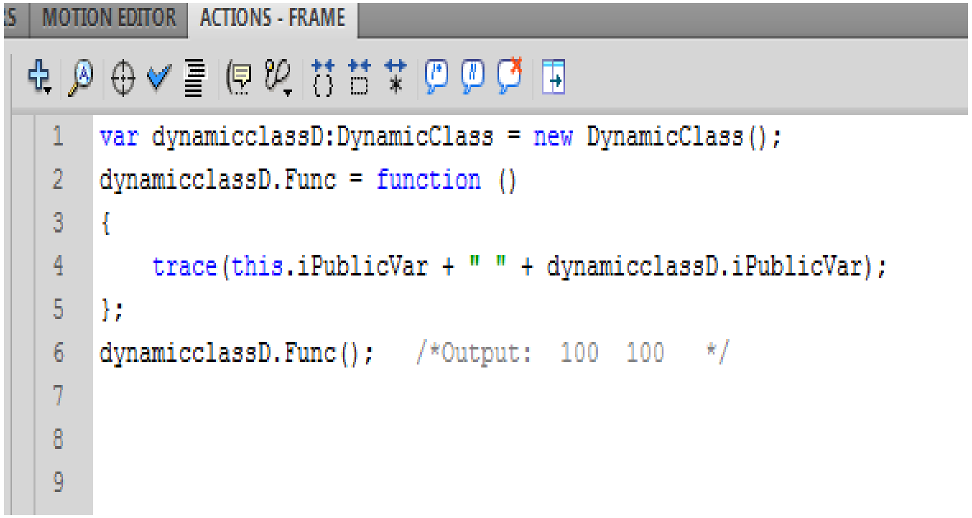
Dynamic Class

- Methods created in this way, however, do not have access to any private properties or methods of the dynamic class

- Moreover, to access the public properties or methods you must use the ***this*** keyword or the class instance name.



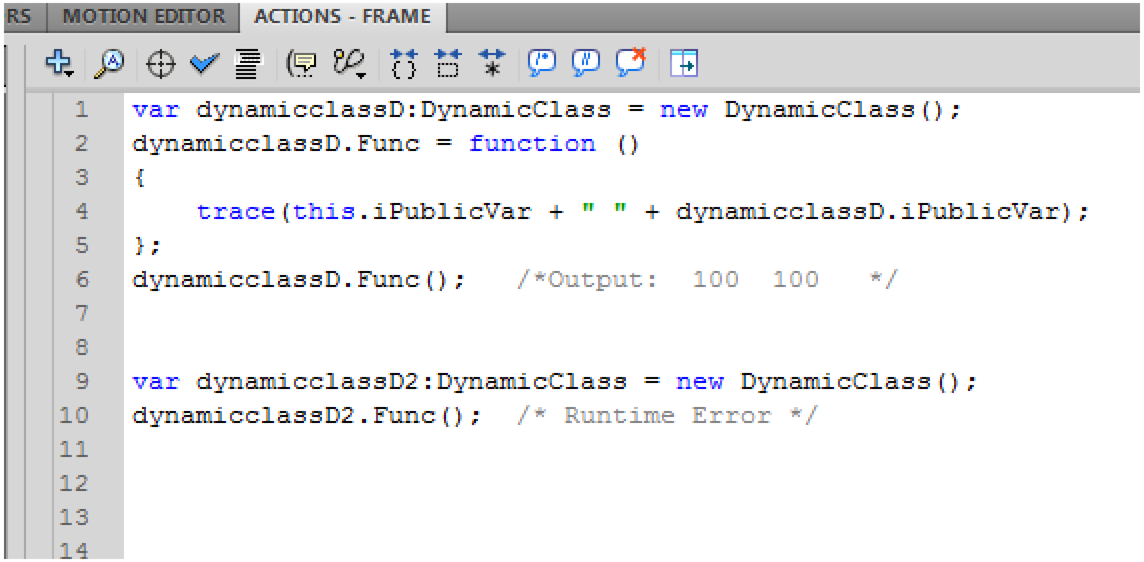
```
package
{
    dynamic public class DynamicClass
    {
        public var iPublicVar:int;
        private var iPrivateVar:int;

        public function DynamicClass()
        {
            iPublicVar = 100;
            iPrivateVar = 200;
        }

    }
}

```



```
var dynamicclassD:DynamicClass = new DynamicClass();
dynamicclassD.Func = function ()
{
    trace(this.iPublicVar + " " + dynamicclassD.iPublicVar);
};
dynamicclassD.Func();    /*Output:  100  100   */



```

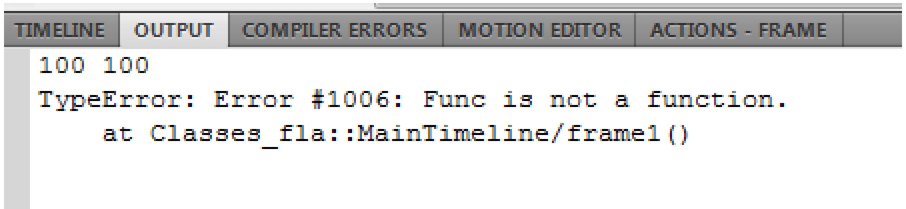
Dynamic Class

- The MOST IMPORTANT thing to understand when adding properties to a dynamic class at runtime is that you are adding those properties to the instance of the class and not the class itself.



```
var dynamicclassD:DynamicClass = new DynamicClass();
dynamicclassD.Func = function ()
{
    trace(this.iPublicVar + " " + dynamicclassD.iPublicVar);
};
dynamicclassD.Func();    /*Output:  100  100   */


var dynamicclassD2:DynamicClass = new DynamicClass();
dynamicclassD2.Func();  /* Runtime Error */
```

The runtime error looks like this:



```
100 100
TypeError: Error #1006: Func is not a function.
    at Classes_fla::MainTimeline/frame1()
```

The End ☺