**PseudoActionScript Serialiser/Deserialiser**

**How To Use the Files**

Extract the zip and paste the Deserialiser folder into your project folder. Assign your FLA document’s classpath to Deserialiser folder.

In your actions panel, or AS file, type

*”*

*import Pasv2. \**

**PseudoActionScript Deserialiser**

**Convert actionscript code into objects ,and perform functions , all during Runtime!**

**The Deserialise method**

Pasv2.Deserialiser.deserialise (codeblock:String, rootObject:Object,rootRef:Object, usePersistent:Boolean);

This function interprets and executes the PseudoActionscript (Pas) code in the string codeblock.

The rootObject refers to the root reference of your swf. Use it to specify the document class.

Alternatively, you can set rootObject to an object that is not the root, but all the objects and values created by the deserialiser will be assigned to that object instead.In this case, if you want to reference some objects outside your rootObject, set rootRef to your document class.

For example,

//If rootObject is root.dataStorageObj;

Then if your Pas code is:

this.object1 = {};

this.ref= root.referredObj;

Deserialiser.deserialise(code,root.dataStorageObj,root);

//object1 will be stored in root.dataStorageObj instead

//ref will refer to the referredObj in your document class of the document class.

You can debug your swf containing the Deserialiser to see the new variables assigned to your swf as a result of your Pas code.

This is a beta version that has not been extensively tested. If you find any bugs, you can report them to

Assholela138@hotmail.com

I hereby release PasD code into the public domain. Distribute and modify as you please.

**Introduction**

Serialise means convert a bunch of complex data, like objects, into a shorter and readable form, like code strings. Deserialise means converting those strings back into data.

The Pseudoactionscript Deserialiser (PasD) inteprets and converts strings employing pseudo Actionscript syntax (Pas) into objects and functions that can be created and called during runtime. Its accompanying serialiser turns actionscript objects into Pas, and is called PasS appropriately. **Unlike Actionscript, Pas is not “compile-once” and can be run during runtime. It can also be used in your swfs themselves to create dynamic code and object structures, such as load/saves.I.e when actionscript compiles from your fla, it is unchangeable. But your Pas code can change and build its own objects when your swf is running!**

**Why use PseudoActionScript**

**1. A simpler and more understandable approach of saving and loading games/software states.**

For example, you could save your game in XML:

*<someGame>*

*<gameInterface>*

*< gameWorld size = "100">*

*<Character life = "100">*

*<type> Hero </type>*

*<gun> pistol </gun>*

*</Character>*

*<enemy1 type = "wolf">*

.....

In large complex programs, your xml document might end up very messy. You have to create your own deserialiser function to make sense of all those tags! And if something goes wrong with the loading/saving ...it would then become very difficult to eyeball where the error is, especially with all that nesting! The <> can also be quite an eyesore. Computer compilers love those but they are meaningless junk to human eyes. And although flash provides quite neat XMLNode Accessing functions, you’ll still have to deserialise it yourself!

ByteArrays are new in Flash and they allow you to serialise stuff and dserialise them on will. However, once they are serialised, ByteCode is meaningless, and imagine trying to scan it for some errors/perculiarities that may pop up. Imagine that your game failed to set the Ai.Timer object of some enemy far far away which you don’t know of. By saving and serialising, you can theoretically use it as a means of visualising the entire game state including any errors. But you can’t do that with byteCode, unless you’re a computer (genius). Also, byteArrays can’t parse parameters.If you create a instance object using new instance (param1,param2,param3), ByteArrays will fail! (Adobe themselves said so!)

**Heres the (P)actionscript alternative:**

*root.gameInterface = new GameInterface();*

*root.gameInterface.gameWorld = new gameWorld();*

*//assign the gameworld to a local variable*

*gameworld = root.gameInterface.gameWorld;*

*gameworld.Character= new Hero ();*

*gameworld. Character.life = 100;*

*gameworld. Character.gun = "pistol";*

*gameworld.enemy1 = new Wolf ();*

*...*

A lot neater, huh, and all the references are laid out properly.And comments are also allowed! (Sure, XML also allows them, but the < --- ! --- > is unwieldy). If you know how to create Actionscript, you'll definately know how to intepret the code.You can also create more complex and meaningful code. For example, in XML, it is impossible to directly say that your character is a Hero, just that your character contains a property called “Hero”. And unlike byteArrays, you can edit the text file containing Pas manually without the need for a “World Builder” Application, or for it to be deserialised so that you can see it visually in Flash.

**2. A more convienient approach of sending commands thru network**

Instead of compressing whatever you want to send into meaningless title and datapacket arrays such as ["walk, hero , 3,3"], you can send in the direct command to use on your hero: "gameworld.Hero.walk(3,3);" And you won’t have to deserialise the packet array youself. PasD executes the code for you!

**3. Runtime Command Prompt sytle Debugging**

Lets say you encounter a bug when certain conditions happen, like your game crashing if your hero steps on a bridge which breaks when 10 enemies are on top of it as well. The probability of that happening is one in 10,000, but in order to fix the bug you must reproduce it. With PasD, you can create a "Command Prompt" in your game. At runtime, you could simply type in some code to shift the enemies on the bridge when you are on it and then call Bridge.Break(); Voila, crash reproduced!

The Adobe Debugger allows you to set/change variables in runtime as well, but thats only restricted to primitive types. It won't allow you to create new objects, classes and such, and

everybody knows that when a game is complex, the Adobe Debugger lags! You also don't need to open flash to recreate conditions for bugs to happen. You can also view object properties by serialising them using PasS and then writing them to your console. A sample console is attached in the zip file.

**4.Flash Compatibility**

That simply put is, Pas is completely similar to Actionscript! If you serialise to Pas, you could simply paste the code into the actions panel and it'll work. Packages, classes, variables and methods native to flash are also available for you to harness!(of course, you must import them into your swf first).For example, you could call mc.addChild(anothermc); using Pas and it’ll work.

**5. Assignments by Reference**

PasD (and actionscript) gets around a very tricky bug that few sedes-es (serialiser deserialisers) out there can solve. Lets say you want to build a hierarchy based object model like a family tree, and each node on the family tree is a object:

grandMa = {};

Mother = {};

grandMa . child = Mother;

Mother.parent = grandma

Most sedes(es) assign by value.Tto serialise the grandma.child property, you’ll need to serialise Mother, and all its properties. But Mother also has a property called parent which refers back to grandma. So you’ll have to serialise grandma again, and all its properties, including child, which is Mother, and so on..... Eventually your code will be infinitely long. With Pas, you can throw direct references , and PasD can interpret those references, so if you ever build an application or game which consists of a tangled web of references here and there, you wont ever need to worry about how complicated your serialised code needs to be.

**What PasD can do:**

**1. Evaluate and assign strings, booleans and numbers**

**2. Evaluate and assign absolute references**

**3. Assign and retrieve the value of local variables**

**4. Get, but not set, Static class variables (in packages)**

Static vars are there for a reason, to be static. Who would want to change them in runtime anyway?

**5. Perform, and retrieve the value of function closures attached to an object, or class instance and static methods.**

**6. Instantiate classes**

**7. Create comments**

Just like in real AS, whenever you include "//" in front of any line in your code, the deserialiser will ignore it. So comment away!

**What PasD can never do**

**1. Perform operations , e.g 3+2 or while(condition), of if ()s and so on...**

**2. Define functions and classes**

**3. Evaluate really crazy expressions like root.foo = new Foo(new bar(3, String('foo')), {x = 5, y = 5});**

it cant nest brackets more than one level

**4. Type variables. e.g someNumber:Number = 3**

variable typing are only meant to generate compiler errors, so they are useless in a runtime setting, don't you agree?

**5. Substitute for REAL actionscript**

PasD is just a simple deserialisation tool. It can only perform simple function calls and assignments. It cannot run an entire program for you!

**Pas Syntax**

**Much of Pas is similar to actionscript. However, there are a few limitations in Pas compared to AS3 that you need to take note of in your syntax.**

**Referencing**

If your document instance contains a object (can be a class instance or whatever) called StuffContainer, and that container further contains another object called StuffSubContainer, and you want to reference a value called stuff inside StuffSubContainer, use

*this.StuffContainer.StuffSubContainer.stuff*

You set the object which will act as (this) by adding its reference to the *rootObject* param of the deserialise function. I called the param *rootObject* because its the ‘root’ of everything that is stored.

References can be nested indefinitely, there is no limit:

For example,

*This .obj1.obj2.obj3.obj4.obj5........ obj100*

still returns you the reference of obj100

You can use that reference on the LHS of an ***"="*** to assign a variable, or on the RHS to assign a reference to a variable, or as a parameter in a function.

You can reference a variable thats not within your storage Object by setting the rootRef param of the deserialise() function to your document class, and then by coding :

*root.ref*

In Pas.

If you want to reference a static class variable, just use the classname if the package is a default or top level package (package { }) e.g

*someClass.staticVar*

If the class is in a package, import the class using

*import somePackage.someClass;*

**N.B You can only import classes individually in Pas. Importing packages with *import package\** throws an error.**

**Also, you can’t import methods or properties of classes. E.g import package.Class.someMethod;**

On top of all your code, and then call

*someClass.someStaticVar;*

Alternatively, you can type in the fully qualified class name, ie. Package.subpackage.class , and include the <pk> tag in behind the classname between the “.staticVariable”.

For example

somePackage.someClass<pk>.someStaticVar

***Note: Dynamic references, i.e the array access operator root[someVarName] is not supported and throws an error.***

**Referencing local variables**

PasD supports local variables. These variables are temporary and callable only when the Pas script runs after which they are deleted.(unless you set the 3rd parameter of deserialise(), usePersistantLocals to true)

Use locals to shorten otherwise unwieldy absolute references and to create complex objects to be passed as arguments or into an array or object in your runtime.

**Any reference that does not have a root. or this. in it is considered local e.g**

*root.object = someValue;*

*//this assigns an object to the swf document class (rootRef)*

*This.object = someValue;*

*//this assigns an object to the storageobject (rootObject)*

*object = someValue;*

*//this assigns an object to the local storage space*

For convention and flash strict typing compatibility, you can append the statement var (“var <Space>”) in front of the assignment. PasD removes and ignores them completely tho.

*var object = someValue;*

*//this assigns an object to the local storage space*

You can assign local vars to references in your runtime environment, primitive values, arrays, and other local references e.g

*localvar = true;*

*localvar = root.object;*

*localvar = [1,3, 'string', true];*

*localvar = anotherlocalvar;*

**N.B :THE SEMICOLON IS ABSOLUTELY VITAL. DO NOT MISS IT! THE DESERIALISER INTEPRETS COMMANDS BY THE SEMICOLON, NOT NEXTLINES, SO IF YOU MISS IT, YOUR COMMAND WILL BE SKIPPED (or Flash will just throw an error and cease to function).**

You can nest and add properties to local objects e.g

*var object = {};*

*object.subobject = {};*

*object.subobject.someValue = "value";*

Of course, you'll have to assign the parent objects first before you can assign the children

For your localobjects to have any use, you'll eventually have to assign runtime objects to local vars

e.g

*someLocalObject = {};*

*someLocalObject.property = "property";*

*this.someObject = someLocalObject;*

**Assigning primitive values to the runtime**

Primitive values are strings, numbers or Booleans, or null. You assign then to variables like in real AS

for Strings

*someref = 'string';*

*or*

*someref = "string";*

for Numbers

*someref = (number);*

* *root.number = 3;*

for Booleans

*someref = (true or false);*

* *root.boolean = false;*

for null

*someref = null;*

*Never set someref = undefined.*

*It will presently throw an error in Flash. In PasD, undefined means an error has happened parsing the value, and error catches will be done to take care of them.*

assigning a reference to null effectively deletes the reference.

If PasD cant parse an expression, it will return an error.

**Preferably, add a space before and after the equals. Although it may still work, I don't know and can't guarantee extensively what would happen if you miss a space or two, or add too many.**

***If you want PasD to support any other primitive datatype such as XML, email me and I’ll update it when I’m not engaged in any other project. My current project is an RTS game, I built PasD just so that it can load/save and be multiplayer.BTW, I hate the way XML looks and also the intangible structure of ByteArrays!***

**Assigning Arrays**

*someref = [element1,element2,element3...]*

**N.B the element identifier cannot contain another "[", "]" or ","**

which means [ [1,2,3],someFunction(1,3)] is illegal. if you want complex objects in the array, assign them to locals and then pass the reference . e.g

*Objelement1 = {};*

*Objelement.x = 3;*

*Objelement.y =12;*

*FunctionResult = root.someFunction();*

*Arrayelement =['a','b','c','d',functionResult];*

*array = [Objelement, Arrayelement, true,"1"];*

Alternatively, you can use the properties Array.n to manually assign elements line by line e.g

*array.0 = someValue;*

*array.1 = someValue2;*

...

**N.B if you paste this into the actions panel, it will throw an error!**

**Assigning Objects**

Currently, PasD can only assign blank objects based on their literal value {}. To create complex Objects, assign a local to {}, then manually add properties to it.

Assigning to new Object() works as well.

e.g

*localObject ={};*

*localObject.property1= "prop";*

*localObject.nestedObject ={};*

*root.complexObject = localObject;*

Alternatively,

*root.complexObject = {}*

*localObject = root.complexObject;*

*localObject.property1= "prop";*

*localObject.nestedObject ={};*

**Create new instances of classes**

To create class members, first see if they are imported in your ActionScript code. PasD cannot import classes itself (the Pas “import” statement doesn’t do anything special, it just tells PasD that MovieClip is really flash.display.MovieClip). **Classes will only be compiled in your swf if a line in your as code references it, like instance = new myClass(); or myClass.staticFunction, so make sure your class is actually compiled before you reference it using PasD!**

You can then instantiate new class instances in runtime using Pas :

If the class is part of a package, include:

*import someclassPackage.someClass;*

On top of all your other code,

then, to instantiate, you MUST CALL THE CLASS BY ITS SHORTHAND NAME, i.e

*someref = new someClass (params);*

*not*

*someref = new someclassPackage.someClass (params);*

*//throws an error*

PasD supports up to 10 parameters for class constructors. If you want to include more, follow the instructions in the next page.

**N.B if you do not import packaged classes or use <pk>, then PasD assumes the qualified name is a local reference, and searches for it instead.**

**Known error: Your package name cannot be the same as your class name, i.e Foo.Foo is illegal.**

Put your parameters exactly as how you would type then in REAL AS in the brackets. **But do not include complex expressions like elements with additional commas and brackets. If you want complex objects in the params, assign them first to locals and then pass the localvar’s reference.**

If your class constructor handles more than 10 parameters:

**If your class is custom:**

**Create a public static function in your class definition called "createInstance" (MUST BE EXACTLY THE SAME, CAPS AND ALL).**

**The createInstance must accept all the parameters used to create the class, and pass them into new [ClassName] (classname is the name of your custom class).**

*Class ClassName{*

*public static function createInstance (...rest){*

*var instance = new ClassName (rest[0],rest[1].....,rest[n]);*

***return instance;***

*}*

or

*Class ClassName {*

*public static function createInstance (arg1,arg2,arg3,...,argn){*

*var instance = new ClassName (arg1,arg2,arg3,...,argn);*

***return instance;***

*}*

**If the class is built in,**

**create a createInstance function somewhere in your fla code and assign builtInClass.prototype.createInstance to it. Of course, the prototype assignment must come before you plan to execute your Pas.**

*import somePackage.subPackage.someBuiltInClass;*

*someBuiltInClass .prototype. createInstance =function (...rest) {*

*var instance = new ClassName (rest[0],rest[1].....,rest[n]);*

*return instance;*

*}*

N.B Assigning functions to prototypes may cause some things to malfunction in Flash. For example, if you assign prototype functions to strings and then add a string to a textfield using TF.appendText, flash throws and error! (Don’t know why).

**Calling functions**

Call functions using the references to them and including the parenthesis () behind, just like in real AS. Include your parameters in the parenthesis.

e.g

*root.someFunctionContainer.someFunction();*

*root.someFunctionContainer.someFunction(param1, param2,param3);*

if your function has a return type, you can assign functions to variables using Pas. **(if the function is void or doesn’t return anything, Flash may throw an error, so be careful)**

*root.someFunctionResult = root.someFunctionContainer.someFunction();*

*localvar = root.someFunctionContainer.someFunction();*

**Make sure to include the (). Otherwise, just like in flash, the variable is assigned to the Function object instead. (then, you can call the function from that reference)**

**e.g**

***localvar = root.someFunctionContainer.someFunction;***

***localvar();***

You can call static functions (and assign their return values to variables) by referencing the classname containing the static function. If the class is in a package, be sure to import first!

e.g

*StaticClass.staticFunction();*

*import StaticPackage.staticClass;*

*staticClass.staticFunction();*

**Creating Comments**

Use comments to describe what this section of code or serialised data is about.

At present, comments can only be created by line using the “//” before the comment. /\* and \*/ throws errors.

Use multiple lines of “//” to create multiline comments.

Like the AS compiler, PasD ignores anything that comes in the same line after a “//”.

e.g

*// your comment here*

*//another line of comments*

**N.B to prevent PasD from time outing after 15s, a maximum of 1 million lines of comments are allowed. But thats by far more than you’ll ever need, right?**

**Advanced Tidbits**

Because when you save games, the serialised code may not compile in the order the objects were created, references could be assigned in prior commands to objects that haven’t been deserialised, and your deserialisation fails terribly. However, PasD includes a workaround to this problem:

*root.hero = new Hero();*

*root.hero.currentTarget = root.enemy236;*

*stuff;*

*morestuff;*

*....*

*....*

*....*

*...*

*...*

*root.enemy236 = new enemy();*

The hero.currentTarget would reference the enemy236 even before the enemy236 object is deserialised and loaded into the game! And the currentTarget would return null.

Whenever it finds a reference to a target that does not exist in the swf (yet), PasD cleverly searches the entire codeblock for command which assigns the referenced object and then evaluates it, before returning to the current line.

This is what PasD actually does

*root.hero = new Hero();*

*//Error! root.enemy236 does not exist, search for root.enemy236 = something;*

*//found:root.enemy236 = new enemy();*

*root.enemy236 = new enemy();*

*root.hero.currentTarget = root.enemy236;*

*//remove the root.enemy236 = new enemy(); line from further down the codeblock*

*//to prevent duplicate creation*

*//now the reference works!*

*stuff*

*morestuff*

*....*

*....*

*....*

*...*

*...*

**Of course, if the command to create enemy236 does not exist, then PasD will simply assign the Hero.currentTarget to null.**

In addition, if the parent object of something being referenced is not created yet, PasD will create the parent object, the parent parent object, and so on and then assign the parent parent object, parent object and so on and then eventually assign the value of the reference.

E.g. if your serialised code creates the sidebar before the gameMap, and the sidebar referenced a bag of coins in a troll character in a house on the gameMap,

*root.sidebar =new SideBar();*

*root.sidebar.currentItem = root.map.house123.troll.moneybag;*

*stuff;*

*morestuff;*

*....*

*...*

*root.map = new Map();*

*somemorestuff;*

*...*

*...*

*root.map.house123 = new House();*

*...*

*...*

*...*

*root.map.house.troll = new Troll();*

*root.map.house.troll.moneybag = "$3"*

**PasD will do:**

*root.sidebar =new SideBar();*

*//root.map.house123.troll.moneybag does not exist*

*//create root.map.house123.troll.moneybag*

*//root.map.house123.troll does not exist*

*//root.map.house123 does not exist*

*//root.map does not exist*

*//root exists*

*root.map = new Map();*

*root.map.house123 = new House();*

*root.map.house123.troll = new Troll();*

*root.map.house123.troll.moneybag = "$3"*

*stuff;*

*morestuff;*

*....*

*...*

*somemorestuff;*

*...*

*...*

*...*

*...*

*...*

If PasD cannot find a command to create a parent object anywhere in the chain, the entire operation fails and the children of the parent object are not created. They will return null. So as far as possible, try to code for objects top down, i.e the parents first, then the children.

*In Future versions, I will add a functioning WatchList to the deserialiser. All references that haven’t been created will add the corresponding offending line of code into the WatchList. Every time a new reference is parsed, the WatchList checks if the reference is now available. If it does, the line of code is then parsed, and removed from the watchlist. Even when the entire codeblock has finished, PasD then parses the WatchList to sort out any remaining dead References. Eventually when the watchlist cannot be shortened any further, then PasD assumes that that reference really is Dead, and throws an error. This ensures that the order of the code in which you serialise stuff does not matter any more!*

Handling Errors with PasD

When a problem is encountered with PasD, PasD will write the error to its errorLog. The deserialise() function returns a list of errors encountered when parsing your Pas.

Example [AS]:

*var pas = ”var someClass = new ClassWhichDoesNotExistInYourSWF();*

*var someRef = this.deadReference; ”*

*var error = Deserialiser.deserialise(pas,Object(root),Object(root));*

*trace(error);*

*//Error: The Class ClassWhichDoesNotExistInYourSWF does not exist!*

*Line : var someClass = new ClassWhichDoesNotExistInYourSWF();*

*//Error: The Reference this.deadReference does not exist!*

*Line : var someRef = this.deadReference*

PseudoActionScript Serialisation

**Convert your flash objects into ActionScript!**

With Pas package now comes a class called serialisableObject. It basically describes a class which can have itself and all of its properties serialised into Pas.

How to Use the Files

Like Deserialiser, serialisableObject.as is found in the Pas zip file. Unzip the Pasv2 file and place it in your project folder. Add a classpath to the folder. Now you can use serialisableObject class

Include them into your actionscript by using the 2 lines:

*Import Pasv2.serialisableObject;*

*Import Pasv2.feederObject*

Introduction

serialisableObject is the complement to Deserialiser. It converts objects in ActionScript into “PseudoActionScript” code. Which is basically actionscript. When you want to save the game state, for example, you can serialise it into Pseudoactionscript and save it in a text file. You can then reload it using PasD, or paste it into the actions panel, and recreate the game state.

Using the serialisableObject class

The serialisableObject class is basically a abstract class which implements functions for serialising to actionscript. To use serialisableObject, extend it using your own classes.

e.g

*public class myClass extends Pasv2.serialisableObject*

your class myClass now inherits serialisableObject’s serialise function, as well as a few properties which are important in the serialisation process, I’ll get to them in a bit.

You then make serialisableObject a subclass of any built in or custom classes for serialisableObject to inherit its functionality as well. Simply edit the serialisableObject.as file and change the line:

*Public class serialisableObject extends* ***Object***

***Into***

*Public class serialisableObject extends* ***BaseClass***

For example, if you want your character class to be a movieClip and be serialisable, use:

*Public class serialisableObject extends MovieClip*

And then

*Public class character extends Pasv2.serialisableObject*

To have serialisable object inherit from multiple unrelated classes, copy the serialisableObject.as file and rename it as another class, e.g serialisableObject2. Find and replace all references to “serialisableObject” (static functions) into “serialisableObject2” in the code body.

Rename the classfile appropriately. And you now have a carbon copy of serialisableObject that is now another class.

Now you can have serialisableObject2 extend another class like Sprite, and have your own classes extend serialisableObject2.

SerialisableObject has its own assortment of public static functions, but you should treat them as private (the only reason why they’re public is the Object.prototype.serialise method in the serialiseObject.as folder needs to access them as well).

You’re only concerned with the serialise function of serialisableObject, and the static serialiseObject() function. The serialise() function serialises your serialisableObject, and all its variables. You call serialiseObject() on objects that are not serialisableObjects, such as Sprites.

SerialisableObject class Properties

A serialisable object has 3 properties which you must override in your subclasse. The paramsArray, dynamicVarNames(Array) and absoluteRefString. There is a forth optional property called customcomments (String).

**ParamsArray**

The paramsArray contains the **value or reference** of the parameters used to contruct your serialisableObject. For example, if your class Person extends serialisableObject, and you call constructor :

*//Person (name,height, weight, speaksEnglish ,Father, Mother)*

*Var father = new Person(....);*

*Var mother = new Person(....);*

*Var tom = new Person(“tom”,178, 75,false,father,mother);*

The paramsArray contains the values [“tom”,178, 75, false, {Person: father}\*,{Person:mother}];

\*{Person:father} is a reference to father, not a clone. If father is changed, the paramsArray father object is changed as well.

The ability to include parameters in your constructor gives PasS a big pro over the byteArray registerClassAlias () function where class constructors cannot use parameters. Many classes in AS3 use parameters, and its more convenient for you to include parameters into your own class definitions than not. Of course, parameters usually correspond to properties in your new instance, thus you can manually set instance properties thereafter instead of using parameters, but this is not always the case:

*//A bank holds account information, as well as the amount of assets it has, which is the sum of //all the money it has in all its accounts*

*Public class Bank{*

*public var Assets:Number =0;*

*public var accounts ={};*

*public function Bank(){*

*}*

*}*

*Public class Account {*

*Public function Account (bank:Bank, name:String, value:Number){*

*//creates a new account and stores it in the bank referenced in ‘bank’ parameter*

*bank.accounts[name] = this;*

*bank.assets+=value;*

*}*

*}*

Because they update a property of another class instance, it is impossible to omit any parameters. Therefore, it would be nearly impossible to deserialise an account from a byteArray and update the Bank object at the same time, without touching the Bank object itself.

It is not possible to declare parameters in XML or JSON as well.

**To fill in your paramsArray, you reference the serialisableObject constructor by calling super(), and including all your params in the same order as your subclass constructor.**

E.g

*Public class Person extends Pasv2.serialisableObject {*

*Public function Person (name, height,weight, speaksEnglish,Father, Mother){*

***Super(name, height,weight, speaksEnglish, Father, Mother);***

*//yourcode ....*

*}*

*}*

If another class extends serialisableObject, and you want to call the serialisableObject constructor:

E.g

Public class animatedObject extends Pasv2.serialisableObject{

//...

//...

}

*Public class Person extends animatedObject {*

*Public function Person (name, height,weight, speaksEnglish,Father, Mother){*

***super.Super(name, height,weight, speaksEnglish, Father, Mother);***

*//yourcode ....*

*}*

*}*

Invoke super. as many times as you have classes in between the inheritance tree between your subclass and serialisableObject.

When you serialise an object, the paramsArray will be serialised into the brackets of your constructor function:

this.tom = new Person**(“tom”,178, 75, false, this.father\*,this.mother\*);**

**\*When objects are included as parameters, the serialiser can either interpret them as references to other objects in your swf or as separate objects contained within the constructor function. But more on that later.**

If your parameters correspond to properties within your serialisableObject, you can dynamically modify them at will.

E.g

*//Unit (name, unittype,x,y);*

*Var infantry1= new unit(“infantry1”,”ranger”,100,200);*

*//paramsArray will be [“infantry1”,”ranger”,100,200];*

Then when your unit moves, x and y changes, you can update them:

**Using setter functions:**

*function set X (value){*

*this.x = value;*

*this.paramsArray[2] = value;*

*// 2 is the index of x*

*}*

*function set Y (value){*

*this.y = value;*

*this.paramsArray[3] = value;*

*//3 is the index of y*

*}*

**Or manually updating them periodically**

*Function refresh(){*

*//called every frameto update the unit*

*paramsArray[2] = this.x;*

*paramsArray[3]=this.y;*

*}*

So the next time the unit is deserialised and created, it will be at the new X and Y locations.

DynamicVarNames

dynamicVarNames is an array containing the **names, (not references)** to any properties that might change after the object is created.

You manually fill in the names of these properties, and they will be included in the Pas the serialise function produces. **Properties which are not included in the dynamicVarNames will not be serialised.**

Using the unit example again:

*//Unit (name, unittype,x,y);*

*Class Unit extends Pasv2.serialisableObject{*

*Public function Unit(name, unittype, x, y){*

*Super (name,unittype,x,y);*

*//update the paramsArray;*

***this.dynamicVarNames = [“x”,”y”];***

*}*

*}*

You need only fill in the names of the properties which change during runtime (hence the word dynamic). In this example, the unitname and unittype are fixed, a unit does not change its type after it is created, so it is not necessary to include them. If the property is a reference and the object changes, you’ll have to include it in dynamicVarNames as well, even though the reference itself points to the same object.

There may be also a whole bunch of variables in your instance that do not need to be serialised. For example, you can set a functional property “isInWater” everytime your unit moves, to be true whenever your unit is crossing a river, but your move function can figure this property out based on its x and y coordinates, so it is unnecessary to save this property.

In addition, all dynamic properties of a dynamic class are serialised by default even if they are not included in dynamicVarNames. For example, if you define a property “prop” on the dynamic Class MovieClip, it will always be serialised.

**N.B private class properties can never be accessed externally, so they can never be serialised. Set them to protected or public if you want to serialise them.**

AbsoluteRefString and Absolute References

In the context of object:

*this.object ={};*

*object.childobject ={};*

***this.object2= {};***

***object.childref= this.object2;***

Childref and childobject are treated as similar by Flash. They are both references contained within object, and flash has no way of telling which is a child (or property) of object or just a reference to another object. Although in this case it is clear that childObject is a property and childref is a reference, Actionscript code cannot tell for sure, and if you write your own serialisation code, you’ll end up with a conundrum over how to deal with such cases.

Ideally, you want children to be serialised as:

this.object.property = **new propertyClass (...,...);**

and you want references to be serialised as:

this.object.ref = **this.someRef;**

If you treat both childObj and childRef as children, you’ll end up with a clone of this.object2 in this.object.childref. Then childref will have absolutely nothing to do with this.object2, and if object2 changes, childref remains the same. Not only do you waste memory, you’ll foul up your application as well! Of course, you don’t want to serialise by reference only, then nothing gets created!

In addition, if someRef.ref references back to this.object, then you have an infinite loop; you’ll serialise someRef, then you’ll serialise the property ref, which is object, and then you’ll serialise the object.ref as well, which is someRef, and so on until your app crashes (or your serialised code is a gazillion bytes long).

On top of all that, what if you want to reference an external object that is not serialised? For example, you have a gameMap with all its units. Each unit has a reference to Game.multiplayer which allows them to interact with your game multiplayer console. But when you start a new multiplayer game and load a map,you already have a multiplayer console; you absolutely don’t want to load your multiplayer object a hundred times, that will cause your connection structure to go haywire!

A lot of serialisers, JSON, XML, ByteArray... serialise by value (i.e treat references as children), and get tripped up by all these nitty gritty nuances. But PasD handles references in the form of string paths, and gets past these traps. Hurray for Pas

This is where absoluteRefString comes in. A serialisableObject stores a string which refers to the absolute path of itself, so that references to it in thru its absolute path will be cause it to be serialised and other references will be serialised as references. SerialisableObjects have this property by default, if you have dynamic objects, you can define the property “absoluteRefString” in runtime as well.

Heres how PasS handles the following data structure:

**[AS]**

*This.object = new serialisableObject();*

*//serialisableObject contains absoluteRefString, which you set yourself*

***This.object.absoluteRefString =”this.object”;***

*This.paramRef = {};*

*//{} is a dynamic object*

***This.paramRef.absoluteRefString =”this.paramRef;”***

*This.paramObj ={};*

*paramObj.prop =”prop”;*

*Var anotherObj=new serialisableObject(this.paramRef,this.paramObj, {x:3,y:3});*

*anotherObj.childObject = new serialisableObject();*

*anotherObj.ref = this.object;*

*//serialise anotherObj*

*var string = anotherObj.serialise(....);*

**[Pas]**

*Import Pasv2.serialisableObject*

*//Pas creates local param Objects first*

*Var A\_arg\_1 =* ***new Object();***

*A\_arg\_1.prop =’prop’;*

*Var A\_arg\_2 =* ***new Object();***

*A\_arg\_2.x=3;*

*A\_arg\_2.y=3;*

*this.anotherObj = new serialisableObject(****this.paramRef****,A\_arg\_1,A\_arg\_2);*

*this.anotherObj.childObject=* ***new serialisableObject();***

*this.anotherObj.ref=* ***this.object;***

For references with absoluteRefString defined, the absoluteRefString gets substituited for the serialised definition of the properties instead. This goes for parameters as well. Parameters with absoluteRefString defined will get it substituited for the definition, otherwise, PasS creates local param Objects A\_arg\_x to represent them. Note that although this.paramObj resides on this, its absoluteRefString is un defined and so is treated as a “child” parameter.

By default, if absoluteRefString is not defined, the property is treated as a child. But PasS has a further 2 tricks up its sleeve to prevent those nitty gritty nuances from happening:

1. SerialisableObject contains a static refsContainer dictionary that stores the absoluteRefString of any object you specify. The refsContainer is for non dynamic, non serialisableObjects to store an absoluteReference, since the property cannot be stored on themselves. Since refsContainer is a dictionary, you specify the reference of the object as the key and then the absoluteReference of that object as a value:

[AS]

*This.refedObject = new Sprite();*

*//sprite cant have absoluteReference as a property, store it in refscontainer*

*SerialisableObject.refsContainer[this.refedObject] = “this.refedObject”;*

*//now PasS will use its reference instead of serialising the member*

2. When serialising stuff, PasS stores the reference of objects it has already serialised into its own IteratedObjects dictionary, so that the next time it encounters the same object, it will use the absoluteReference of its serialised form instead of creating the object itself:

[AS}

*This.cont={};*

*This.cont.objectReffed={};*

*This.cont.object1={};*

*This.cont.object1.ref = this.objectReffed;*

*This.cont.serialise(....,...);*

//Pas

//the order of serialisation does reflect the order in which objects are created

*This.cont = new Object();*

*This.cont.object1= new Object();*

***This.cont.object1.ref*** *=* ***new Object();***

*This.cont.objectReffed =* ***this.cont.object1.ref;***

***Although this provides a failsafe from the recursive reference conundrum, notice that the reference structure gets messed up. Therefore, always use AbsoluteRefStrings and refsCOntainer when possible.***

Custom Comments

Custom comments are where you describe your serialised data. It is a string, and acts like comments in actionscript. Comments are stored in the customcomments property of the serialisableObject class. You set it to whatever you want to describe the object.

For example:

*Class stuff extends Pasv2.serialisableObject{*

*Public function stuff(params){*

*Super(params);*

***this.customcomments = “// Just some stuff”;***

*}*

When it is serialised, the customcomments will appear before the code to generate the object, as well as any other parameters the object needs:

***//Just some stuff***

*...< params Init>.*

*root.someObject = new Stuff(params);*

*...<dynamic Properties init>...*

*....*

Remember to put // in front of your comments. Otherwise, PasD will deserialise your comments and may throw an error!

The serialise Function

Now that you’ve got all the serialisableObject structure in place, you can call the serialise function to serialise your serialisableObject into pseudoActionScript. The serialise() function generates a string which represents the serialised form of the object, based on your paramsArray, dynamicVarNames, absoluteRefString and customcomments. All dynamic variables and properties you specify in dynamicVarNames will be serialised as well.

Serialise accepts a few params

Serialise (referenceStr, includeVar, isTopParam,thisObject, feederObject);

**referenceStr:String, this is a required parameter:** The reference you want your serialisableObject to be serialised to.

E.g

Var string = This.serialise**(“this.reference”);**

string:

**this.reference** = new serialisableObject();

Root.reference.prop1 = someproperty;

...

N.B, the this. In Pas refers to rootObject parameter when you call Deserialiser.deserialise.

**includeVar: Boolean, default false:** Whether there is a var in front of the object reference. Its of no use with PasD, since it ignores var statements, but should you want to put your serialised code into the Actions panel, the complier will throw an error if you do not var a local variable. Set this to true in that case.

Var string = This.serialise**(“localvar”, true);**

string:

**var localvar**  = new serialisableObject();

localvar.prop1 = someproperty;

...

**isTopParam:Boolean, default = false:** Because serialise() recursively calls itself for every property in the serialisableObject, the inTopParam decides whether it is YOU who have called the serialise() function or the code. **Set this to true**. When isTopParam is set to true, the serialisableObject. globalobj which contains the references of local vars, iteratedReferences etc is reset and initialised. If it is false, it is untouched.

You can set it to false, but then you may have local variables left behind from previous calls to serialise , and conflicts in the iteratedReferences.

The only time you want to set this to false is when you code for asynchronous saving, (saving in a period of time as opposed to one frame). Setting this to false will allow you to reuse localvars and iteratedReferences from serialise() calls every frame or so when you perform asynchronous saving. More on asynchronous saving/loading with Pas is described later.

**thisObject:Object , default = null:** This is a deprecated parameter, which specifies another object for a serialisableObject to serialise() on its behalf. It has been superceded by the static serialiseObject() function and is no longer of use. **Set this to null.**

**feederObject:Pasv2.feederObject , default = null:** If you want the object to be serialised according a custom set of paramsArray, dynamicVarNames and absoluteRefString that is not stored in the serialisableObject itself, pass a Pasv2.feederObject into this param. More on feederObjects in the later section.

**Serialising Non serialisableObjects**

Although you can make virtually any custom class serialisable by using serialisableObject to extend it and then extending serialisableObject to your own classes, sometimes its more convienient to use built in classes, e.g Points and Objects directly. PasS can also serialise non serialisableObjects, but the approach is more tricky because you cant define paramsArray, dynamicVarNames etc on the instances themselves. There’s a workaround to this problem.

You can serialise any object by calling the serialisableObject.serialiseObject function.

The arguments of serialiseObject is similar to serialise() , except for 2 additional parameters:

serialiseObject(**thisObject**,referenceStr,includeVar,isTopParam,**feederObject**) ;

**thisObject: Object, this is a required parameter**: The object you want to serialise. Put the object reference itself here, not any reference String. Then, referenceStr specifies the reference of the thisObject you want to serialise to.

**feederObject: Pasv2.feederObject, default = null:** In serialise(),feederObject is trivial, but in the case of non serialisableObjects, it is absolutely crucial because the object itself cannot have any serialisableObject properties. So the feederObject specifies them for the object.

Using FeederObjects

A feederObject is a special class in Pasv2 that contains the 3 properties, paramsArray, dynamicvarNames and customcomments, just like serialisableObjects, but it cannot be serialised. You use them to specify the serialisableObject properties on behalf of the non-serialisableObject you want to serialise. FeederObjects don’t specify absoluteRefStrings because the referenceStr param in the serialiseObject () function specifies it already, and you can store absoluteReferences in refsContainer.

FeederObjects are created using:

new FeederObject(paramsArray, dynamicVarNames, customComments);

The array you put in paramsArray parameter will correspond to the paramsArray in feederObject, the dynamicVarNames array with dynamicVarNames and customComments with customcomments. Set these properties youself and then create the feederObject using these properties. You can also create the feederObject using (null,null,null) values and then set the properties thereafter.

Example:

Lets say you want to serialise a point:

root.point= new Point(**100,200**);//**(x,y)**

// sometime in your game, the point’s coordinates are changed

root.point.x = **130;**

root.point.y=**550**;

//To serialise:

// specify the params of the point

var pointParams =[**100,200**];

//specify the dynamicVarsNames of the point. Point has 2 properties which change, x and y.

Var dynamicVars= **[“x”,”y”];**

//create custom comments if you want

Var cc = **“// A point in root”;**

// create a feederObjects to specify the serialisable properties of the point

Var pointFeeder:Pasv2.feederObject = new FeederObject(**pointParams,dynamicVars,cc**);

// now ,serialise the point and add the feederObject into the feederObject param

serialisableObject.serialiseObject(root.point,”root.point”,false,true,null,**pointFeeder**);

serialised Code:

*import flash.geom.Point*

***// A point in root***

*root.point = new Point (****100, 200****);*

*root.point.x =* ***130;***

*root.point.y =* ***550;***

So you see, the parameters you put in feederObject.paramsArray appear in the brackets, the dynamic Vars x and y which have changed are reflected in the serialised code via feederObject.dynamicVarNames, and you can have custom comments put in front of your code.

But what if the object is a property of another serialisableObject, and gets serialised automatically? Then how do you specify the feederObject?

Fortunately, serialisableObject comes with a function called generateFeederObject, which generates feederObjects based on its entry in the feederDictionary. generateFeederObject() provides the feederObjects for properties which are serialised automatically.

The FeederDictionary

For built in classes, there are consistent ways of serialising them. For example, their parameters always correspond to some properties, they always have the same properties which vary dynamically, and there is a consistent way of describing them. Hence, you can put a feederObject into serialisableObject.feederDictionary which represents the serialisableObject properties of each instance of the class. When PasS serialises properties, it looks up the feeder dictionary for that particular class, and passes it into the feederObject parameter.

We’ll use a rectangle example:

//define a consistent feederObject for members of the Rectangle class

Var feederRect= new feederObject(null,null,null);

feederRect.paramsArray = [**“x”,”y”,”width”,”height”];**

**//note that in the paramsArray, you put the names of the variables which correspond to the params, in order, not their values**

feederRect.dynamicVarNames= [“x”,”y”,”width”,”height”];

//same as always, the properties that will change.

feederRect.customcomments = “// A rectangle”

import flash.geom.Rectangle;

//To put a class into the feederDictionary:

serialisableObject.feederDictionary[**flash.geom.Rectangle**] = **feederRect**;

**//the key is the class, the value is the corresponding feederObject**

Object(root).containerObj = {};

Object(root).containerObj.rect = new Rectangle(100,200,50,50);

//then the x and y changes

Object(root).containerObj.rect.x=**50;**

Object(root).containerObj.rect.y=**50;**

**//Alternatively , to register the class feederObject**

serialisableObject.feederDictionary[Object(**root.rect.constructor)**] = feederRect;

//

//serialise containerObj

serialisableObject.serialiseObject(root.containerObj, “root.containerObj);

//serialised string

*Root.containerObj = new Object();*

*Import flash.geom.Rectangle;*

*Root.containerObj.rect = new Rectangle(****50,50,50,50****);*

***Root.containerObj.rect.x = 50;***

***Root.containerObj.rect.y= 50;***

***Root.containerObj.rect.height = 50;***

***Root.containerObj.rect.width =50;***

If the properties corresponding to the params are changed, the params change themselves, but that has no effect in built In classes because params are used to initialise properties anyway.

*In future versions, I will allow the instances themselves to have associated feederObjects in the dictionary, but due to time constraints this is not possible at the moment.*

**Asynchronous Saving/Loading**

By saving/loading all your data all in one serialise()/deserialise() call, If the amount of information to serialise/deserialise is large, the flash player may seize at one frame for a couple of seconds to several minutes. This makes your application/game unresponsive, and may cause users to terminate it. Asynchronous saving/loading is a means which you split the information to serialise/deserialise into multiple blocks, each carried out in its own frame. The information will be saved/loaded in the background while your application continues to respond, thereby ensuring a smooth loading/saving experience.

I will first discuss asynchronous loading because it is more linear and easier to explain:

To perform asynchronous loading (i.e deserialisation), split your Pas codeblock into a number of sizeable chunks. Then, run Deserialiser.deserialise() on each chunk of code using the ENTER\_FRAME event.

An example is described here. Add

*//split the codeBlock into chunks*

Function splitChunk(codeBlock){

codeBlock:String = //your entire serialised code

var commandsArray:Array = codeBlock.split(“;”);

//splits your codeblock into individual commands. Commands will be combined

//into chunks of code of a specific size

var chunksArray:Array =[];

var chunkSize:uint=//someNumber, maybe 1000

//the number of lines of code in each chunk

For (var i =0;i<commandsArray.length;i++){

Var command:String= commandsArray[i];

Var chunkNo:uint= Math.floor(i/chunkSize);

//initialise the chunk if it does not exist

If(chunksArray[chunkNo]==null){

chunksArray[chunkNo]=””;

}

chunksArray[chunkNo]+=”\r”+command;

}

//the index of the chunk currently serialised

This.chunkIndex=0;

}

//define a function to

Function deserialiseChunk(evt:Event){

Var chunk:String= chunksArray[this.chunkIndex];

Pasv2.Deserialiser.deserialise(this,Object(root),**true**);

this.chunkIndex++;

}

This.addEventListener