# JOOP, In and Out

## What is JOOP?

JOOP (pronounced joop) is a Javascript pre-processor that adds some semantics I wished for in Javascript. It adds classes, inheritance, and encapsulation in a syntax that makes it look more like Java and C#. There are ways of doing all of these in plain Javascript, and JOOP just makes it easier by automating the processes required to do them. JOOP is meant to be used alongside regular Javascript, and is not particularly conducive for end-user scripting.

The name JOOP is an acronym for the phrase "Javascript (with) Object Oriented Programming". Some will say that this name is technically incorrect, as Javascript is already an OOP language. To that I say: it's just a name, get over it.

In this document I'll take you through how JOOP works, the semantics which JOOP adds, joopc - the JOOP compiler, and some ways which it can be integrated with Visual Studio and Notepad++.

## JOOP, the Language

The language itself is basically Javascript with some flair. Most of the code you'll be writing is plain old Javascript.

### Try it Yourself

The zip file you got this document from should contain a folder called "TryJoop". In this folder you can try any of the code mentioned below. Simple edit the Test.joop file, and then run compile.bat. This will compile it to the file Test.js, which you can inspect or try in a browser. I'd recommend trying these out as you read them.

### Classes

JOOP adds classes which are defined similarly to C# and Java. You can take the following example from regular Javascript.

var SomeClass = function(foo)  
{  
 this.bar = foo;  
};  
  
SomeClass.protoype.FooFunction = function(foo)  
{  
 this.bar \*= foo;  
};

Here we are defining a class called "SomeClass" with a function called "FooFunction", and a local variable named "bar". In JOOP, this is written like below.

class SomeClass  
{  
 constructor(foo)  
 {  
 this.bar = foo;  
 }  
   
 function FooFunction(foo)  
 {  
 this.bar \*= foo;  
 }  
}

Again, JOOP is basically javascript with some flair. The JOOP code above, when processed, will actually be just about equivalent to the javascript code above it. However, with JOOP, this process of setting the prototype, which can be done many times, is automated. Also, some might argue that the second one is more readable.

You can also define static functions.

class SomeClass  
{  
 static function FooFunction(bar)  
 {  
 DoStuff();  
 }  
}

Which will be converted to something like below.

var SomeClass = function() { };  
  
SomeClass.FooFunction(bar)  
{  
 DoStuff();  
};

### Namespaces

Another semantic that JOOP adds is namespaces. If you wanted to define namespaces in Javascript, you might have to do something like this.

var GSC = GSC || new Object();  
var SomeNamespace = SomeNamespace || new Object();  
  
SomeNamespace.SomeClass = function() { };  
  
GSC.SomeNamespace = SomeNamespace;

This is automated by JOOP, allowing you to write the following instead.

namespace GSC.SomeNamespace  
{  
 class SomeClass  
 {  
 constructor()  
 {  
 }  
 }  
}

Or, optionally...

namespace GSC  
{  
 namespace SomeNamespace  
 {  
 class SomeClass  
 {  
 constructor()  
 {  
 }  
 }  
 }  
}

### Inheritance

Classes just wouldn't be OOP without inheritance. To do this in JOOP, you can use a colon to define another class to inherit from. Let's say you had a class like below.

class BaseClass  
{  
 constructor (foo, bar)  
 {  
 this.foo = foo;  
 this.bar = bar;  
 }  
   
 function Switch()  
 {  
 var foo = this.bar;  
 this.bar = this.foo;  
 this.foo = foo;  
 }  
}

You can inherit from it like this.

class SomeClass : BaseClass  
{  
}

To call the base constructor, you can simply use the base keyword.

class SomeClass : BaseClass  
{  
 constructor()  
 {  
 base(null, someValue);  
 }  
}

You can even override functions.

class SomeClass : BaseClass  
{  
 constructor()  
 {  
 base(null, someValue);  
 }  
   
 function Switch()  
 {  
 // Infinite loop!  
 base.Switch();  
 this.Switch();  
 }  
}

In Javascript, this would be a real pain to write, even with an "extend" function to do a lot of the heavy lifting. JOOP automates all of this for you.

// This function was taken from TypeScript.  
var extend = (function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p]; function \_\_() { this.constructor = d; } \_\_.prototype = b.prototype; d.prototype = new \_\_(); });  
var someValue = null;  
  
var BaseClass = function(foo, bar)  
{  
 this.foo = foo;  
 this.bar = bar;  
};  
  
BaseClass.prototype.Switch = function()  
{  
 var foo = this.bar;  
 this.bar = this.foo;  
 this.foo = foo;  
};  
  
var SomeClass = function()  
{  
 base(null, someValue);  
};  
  
extend(SomeClass, BaseClass);  
  
(function()  
{  
 var base = BaseClass.prototype;  
   
 SomeClass.prototype.Switch = function()  
 {  
 base.Switch.call(this);  
 this.Switch();  
 };  
})();

### Properties

The compiler can generate get\_ and set\_ methods immediately using the "prop" keyword.

class SomeClass  
{  
 prop SomeProp;  
}

This is called an automatic property. It's equivalent to the code below.

SomeClass.prototype.get\_SomeProp = function() { return this.automaticProperty\_SomeProp; };  
SomeClass.prototype.set\_SomeProp = function(value) { this.automaticProperty\_SomeProp = value; };

Of course, you're not limited to automatic properties. You can define the get and set methods like below.

class SomeClass  
{  
 prop SomeProp  
 {  
 get  
 {  
 return this.private.SomeProp;  
 }  
 set  
 {  
 this.private.SomeProp = value;  
 }  
 }  
}

Or just define a read-only method.

class SomeClass  
{  
 prop SomeProp  
 {  
 get  
 {  
 return this.private.SomeProp;  
 }  
 }  
}

### Encapsulation

JOOP doesn't fully support encapsulation, only the encapsulation of variables. You can technically define functions, but that will have to be done in the constructor. You can do this through the "private" variable.

class SomeClass  
{  
 constructor()  
 {  
 this.private.somePrivateVariable = "foo";  
 }  
}

This is roughly equivalent to the Javascript code below.

var privateObj = [];  
  
var SomeClass = function()  
{  
 // Get a private key.  
 this.privateKey = privateObj.push(new Object()) - 1;  
   
 // Store the private variable.  
 privateObj[this.privateKey].somePrivateVariable = "foo";  
};

The "privateObj" variable you see there is contained in every namespace. Technically, if you had another class in the same namespace and knew the private key of another object, you could access these variables. So, again, JOOP doesn't fully support encapsulation.

### Combining Javascript

You can run plain Javascript at the top level of files by using the prog keyword.

class SomeClass  
{  
 function Foo(bar)  
 {  
 }  
}  
  
prog  
{  
 var someInstance = new SomeClass();  
 someInstance.Foo("test");  
}

Everything between the braces following a prog keyword will be treated as plain Javascript. Also, all prog keywords will be executed in the same scope. So, the following will execute without a hitch.

class SomeClass  
{  
 function Foo(bar)  
 {  
 }  
}  
  
prog  
{  
 var someInstance = new SomeClass();  
}  
  
class OtherClass  
{  
}  
  
prog  
{  
 someInstance.Foo("bar");  
}

However, the prog keyword can only be used at the top level. So, the code below will fail.

namespace GSC.SomeNamespace  
{  
 class SomeClass  
 {  
 function Foo(bar)  
 {  
 }  
 }  
   
 prog  
 {  
 var someInstance = new SomeClass();  
 }  
}

## JOOP, the Compiler

JOOPC (pronounced "joop-see") is the name of the JOOP compiler. JOOPC has its own in-built documentation. Just run joopc in your command line, or joopc --help if you're old fashioned. Also, in the Compiler folder, there's a readme file.

## JOOP, for IDEs

There is some level of Visual Studio integration included with JOOP. That is, the **--msbuild** option in the compiler. If you want your projects to build the .joop files contained within them, you can follow these steps:

1. Create a folder called "tools" in the project's root directory.  
2. Copy joopc.exe from the Compiler\ folder in this zip file to the newly created tools\ folder.  
3. Open your project and go to Project > (Project Name) Properties...  
4. Click on Build Events  
5. In the Pre-build event command line: textbox, type in the following:

"$(ProjectDir)tools.exe" -m -d -i "$(ProjectDir) " -a

Now whenever your project builds, the \*.joop files will be compiled and run. If there's any errors, it will appear in the error list. If you'd rather everything goes into a single file, just use this value instead.

"$(ProjectDir)tools\joopc.exe" -m -d -i "$(ProjectDir) " -s "script.js"

This will compile everything to a file called "script.js" in the output folder.

You can also write JOOP code in Notepad++ with syntax highlighting using the user-defined language. To do this, you can follow these steps:

1. Open Notepad++  
2. Go to Language > Define Your Language  
3. Click on Import...  
4. Navigate to the "Notepad++\_UserLang.xml" file in the IDE\ folder in this zip file.

Now you'll be able to select "JOOP" form the Language list.