

THE MAIL OF MODULARITY

Section 1

Namespacing Basics

JAVASCRIPT
BEST PRACTICES

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>
    ***probably a bunch of other HTML code***
    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>
  </body>
</html>
```



Let's say this first script was built by the same author of the html page. It will report the current members of the Hall of Fame...

Later, she asks a colleague to build a short file that will list requirements for Hall of Fame Selection.

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>

    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>

    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
hof = document.getElementById("hof"),
fragment = document.createDocumentFragment(), element;
for(var i = 0, x = list.length; i < x; i++){
  element = document.createElement("li");
  element.appendChild( document.createTextNode( list[i]) );
  fragment.appendChild(element);
}
hof.appendChild(fragment);
```

The first script creates the list of Knights who are in the Hall of Fame, and then adds them to the "hof" unordered list.

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>

    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



HALLOFFAME.JS

```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
hof = document.getElementById("hof"),
fragment = document.createDocumentFragment(), element;
for(var i = 0, x = list.length; i < x; i++){
  element = document.createElement("li");
  element.appendChild( document.createTextNode( list[i]) );
  fragment.appendChild(element);
}
hof.appendChild(fragment);
```

REQUIREMENTS.JS

```
var reqs = ["Cool Kid", "Slayed a Dragon", "Good at Swording"],
list = document.getElementById("reqs"),
fragment = document.createDocumentFragment(), element;
for(var i = 0, x = reqs.length; i < x; i++){
  element = document.createElement("li");
  element.appendChild( document.createTextNode( reqs[i]) );
  fragment.appendChild(element);
}
list.appendChild(fragment);
```

The existing `list` gets overwritten due to hoisting! Is this a big deal?

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>

    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
    hof = document.getElementById("hof"),
    fragment = document.createDocumentFragment(), element;
for(var i = 0, x = list.length; i < x; i++){
  element = document.createElement("li");
  element.appendChild( document.createTextNode( list[i]) );
  fragment.appendChild(element);
}
hof.appendChild(fragment);
```

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>

    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
    hof = document.getElementById("hof"),
    fragment = document.createDocumentFragment(), element;

function displayHOF() {
  for(var i = 0, x = list.length; i < x; i++){
    element = document.createElement("li");
    element.appendChild( document.createTextNode( list[i]) );
    fragment.appendChild(element);
  }
  hof.appendChild(fragment);
}
```

But what if `halloffame.js` only provided a global method for displaying the HOF members... instead of displaying them immediately?

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="displayHOF();" >
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
    hof = document.getElementById("hof"),
    fragment = document.createDocumentFragment(), element;

function displayHOF() {
  for(var i = 0, x = list.length; i < x; i++){
    element = document.createElement("li");
    element.appendChild( document.createTextNode( list[i]) );
    fragment.appendChild(element);
  }
  hof.appendChild(fragment);
}
```

But what if `halloffame.js` only provided a global method for displaying the HOF members... instead of displaying them immediately?

CONFLICTING CODE WITHIN MULTIPLE JS FILES CAN CAUSE OVERWRITES

JavaScript files can often create conflicting global elements, overwriting existing, important data.

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



HALLOFFAME.JS

```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
hof = document.getElementById("hof"),
fragment = document.createDocumentFragment(), element;

function displayHOF() {
  for(var i = 0, x = list.length; i < x; i++){
    element = document.createElement("li");
    element.appendChild( document.createTextNode( list[i]) );
    fragment.appendChild(element);
  }
  hof.appendChild(fragment);
}
```

REQUIREMENTS.JS

```
var reqs = ["Cool Kid", "Slayed a Dragon", "Good at Swording"],
list = document.getElementById("reqs"),
fragment = document.createDocumentFragment(), element;

...
```

On a click, `displayHOF()` looks for a `list` and now finds the wrong one! Namely, the new global created in `requirements.js`.

ENTER THE SIMULATED JAVASCRIPT NAMESPACE

Though not native to JS, the namespace concept can limit global impact and provide data protection.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var list = ["Jar Treen", "Maximo Rarter", "Pol Grist"],
    hof = document.getElementById("hof"),
    fragment = document.createDocumentFragment(), element;

function displayHOF() {
  for(var i = 0, x = list.length; i < x; i++){
    element = document.createElement("li");
    element.appendChild( document.createTextNode( list[i]) );
    fragment.appendChild(element);
  }
  hof.appendChild(fragment);
}
```

ENTER THE SIMULATED JAVASCRIPT NAMESPACE

Though not native to JS, the namespace concept can limit global impact and provide data protection.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>

    <button onclick="displayHOF();">
      List Hall of Fame!
    </button>

    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



ENTER THE SIMULATED JAVASCRIPT NAMESPACE

Though not native to JS, the namespace concept can limit global impact and provide data protection.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
```

```
};
```

The key to creating a namespace is a single global Object, commonly called the “wrapper” for the space.

ENTER THE SIMULATED JAVASCRIPT NAMESPACE

Though not native to JS, the namespace concept can limit global impact and provide data protection.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,

};
```

All of the variables that were formerly declared in the global scope will now be properties of the **HOFMASTER** namespace.

ENTER THE SIMULATED JAVASCRIPT NAMESPACE

Though not native to JS, the namespace concept can limit global impact and provide data protection.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

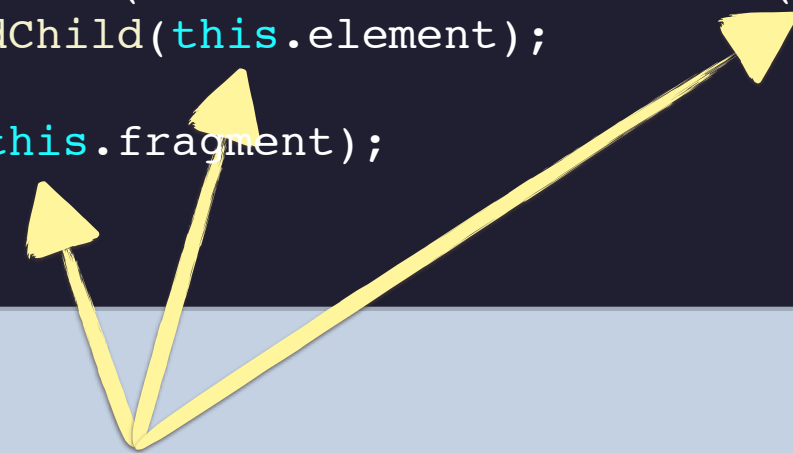
    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
};
```



The `displayHOF` function will also belong to the `HOFMASTER` namespace, which means we'll need to reference the calling object on all the needed variables, using `this`.

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="          ">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
};
```

THE “NAME” OF THE NAMESPACE ACTS AS A SHIELD

Using a mini-environment as a data “container”, we add a layer of protection around important data.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="HOFMASTER.displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
};
```

Now that the function and all the necessary variables are “encapsulated” within the **HOFMASTER** namespace, we’ll need to call that namespace in order to access any of them.

THE “NAME” OF THE NAMESPACE ACTS AS A SHIELD

Using a mini-environment as a data “container”, we add a layer of protection around important data.

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="HOFMASTER.displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../../scripts/requirements.js">
    </script>

  </body>
</html>
```



HALLOFFAME.JS

```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
};
```



REQUIREMENTS.JS

```
var reqs = [ "Cool Kid", "Slayed a Dragon", "Good at Swording"],
  list = document.getElementById("reqs"),
  fragment = document.createDocumentFragment(), element;

...
```

Now, if a globally unfriendly file makes an impact on the document scope, the master information is unaffected.

IDEALLY, ALL INCORPORATED JS FILES WILL USE A NAMESPACE

Namespaces reduce global “footprint” while also keeping data grouped around their intended functionality.

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="HOFMASTER.displayHOF();" >
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../../scripts/requirements.js">
    </script>

  </body>
</html>
```



HALLOFFAME.JS

```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
};
```

REQUIREMENTS.JS

```
var REQUIREMENTS = {

  ***bunch of properties/methods that no longer conflict!***

}
```

If built well, namespaces remains “agnostic” of other namespaces, unless the file-builders design them to work together by providing each with wrapper names.

NESTED NAMESPACING IS FREQUENT IN A MODULE PATTERN

Nesting namespaces provide further organization and protection, as well as help keep variable names intuitive.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="HOFMASTER.displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../scripts/halloffame.js">
    </script>

    <script type="text/javascript"
      src="../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
};
```

NESTED NAMESPACING IS FREQUENT IN A MODULE PATTERN

Nesting namespaces provide further organization and protection, as well as help keep variable names intuitive.

HALLOFFAME.JS

```
<!DOCTYPE html>
<html>

  <body>

    <h1>The King's Hall of Fame</h1>
    <h2>Honorable Members</h2>
    <ul id="hof"></ul>
    <button onclick="HOFMASTER.displayHOF();">
      List Hall of Fame!
    </button>
    <h2>Selection Requirements</h2>
    <ul id="reqs"></ul>

    ***probably a bunch of other HTML code***

    <script type="text/javascript"
      src="../../scripts/halloffame.js">
    </script>
    <script type="text/javascript"
      src="../../scripts/requirements.js">
    </script>

  </body>
</html>
```



```
var HOFMASTER = {
  list: [ "Jar Treen", "Maximo Rarter", "Pol Grist"],
  hof: document.getElementById("hof"),
  fragment: document.createDocumentFragment(),
  element: undefined,
  displayHOF: function () {
    for(var i = 0, x = this.list.length; i<x; i++){
      this.element = document.createElement("li");
      this.element.appendChild(document.createTextNode(this.list[i]));
      this.fragment.appendChild(this.element);
    }
    this.hof.appendChild(this.fragment);
  }
  BIOGRAPHIES: {
    Jar Treen: *some text on Jar*,
    "Maximo Rarter": *some text on Maximo*,
    "Pol Grist": *some text on Pol*,
    list: *some useful list of biography data*
    unfoldBio: function (member){
      *adds text from this[member] to some element*
    }
    *etc*
  }
};
```

A new layer of scope groups related data but shields it by requiring namespace references.

```
HOFMASTER.BIOGRAPHIES.unfoldBio(HOFMASTER.list[1]);
```


THE MAIL OF MODULARITY

Section 1

Namespacing Basics

JAVASCRIPT
BEST PRACTICES

THE MAIL OF MODULARITY

Section 2

Anonymous Closures


JAVASCRIPT

BEST PRACTICES

SO FAR, WE'VE SEEN A MODULE THAT HAS ONLY PUBLIC PROPERTIES

The thing with a namespace is that you have to “hope” that no one else ever uses its name.

```
var ARMORY = {  
  
  weaponList: [ *list of weapon Objects* ],  
  armorList: [ *list of armor Objects* ],  
  
  makeWeaponRequest: function(...){},  
  makeArmorRequest: function(...){},  
  
  removeWeapon: function(...){},  
  replaceWeapon: function(...){},  
  removeArmor: function(...){},  
  replaceArmor: function(...){}  
  
};
```




This namespace has a bunch of public variables and methods which can still be accessed by any code that knows the name of the space and its properties.

WHAT IF WE WANTED SOME STUFF TO BE ACCESSIBLE ONLY TO THE MODULE?

First, we need to decide which properties should be public and which should be private.

```
var ARMORY = {  
  
  weaponList: [ *list of weapon Objects* ],  
  armorList: [ *list of armor Objects* ],  
  
  makeWeaponRequest: function(...){},  
  makeArmorRequest: function(...){},  
  
  removeWeapon: function(...){},  
  replaceWeapon: function(...){},  
  removeArmor: function(...){},  
  replaceArmor: function(...){}  
  
};
```



Since an accurate list of weapons and armor is highly important to accurate armory service, these arrays should be private.

WHAT IF WE WANTED SOME STUFF TO BE ACCESSIBLE ONLY TO THE MODULE?

First, we need to decide which properties should be public and which should be private.

```
var ARMORY = {  
  
  weaponList: [ *list of weapon Objects* ],  
  armorList: [ *list of armor Objects* ],  
  
  makeWeaponRequest: function(...){},  
  makeArmorRequest: function(...){},  
  
  removeWeapon: function(...){},  
  replaceWeapon: function(...){},  
  removeArmor: function(...){},  
  replaceArmor: function(...){}  
  
};
```




The reason an armory exists is to allow folks to get weapons and armor out in orderly fashion, so these request methods should be public.

WHAT IF WE WANTED SOME STUFF TO BE ACCESSIBLE ONLY TO THE MODULE?

First, we need to decide which properties should be public and which should be private.

```
var ARMORY = {  
  
  weaponList: [ *list of weapon Objects* ],  
  armorList: [ *list of armor Objects* ],  
  
  makeWeaponRequest: function(...){},  
  makeArmorRequest: function(...){},  
  
  removeWeapon: function(...){},  
  replaceWeapon: function(...){},  
  removeArmor: function(...){},  
  replaceArmor: function(...){}  
  
};
```

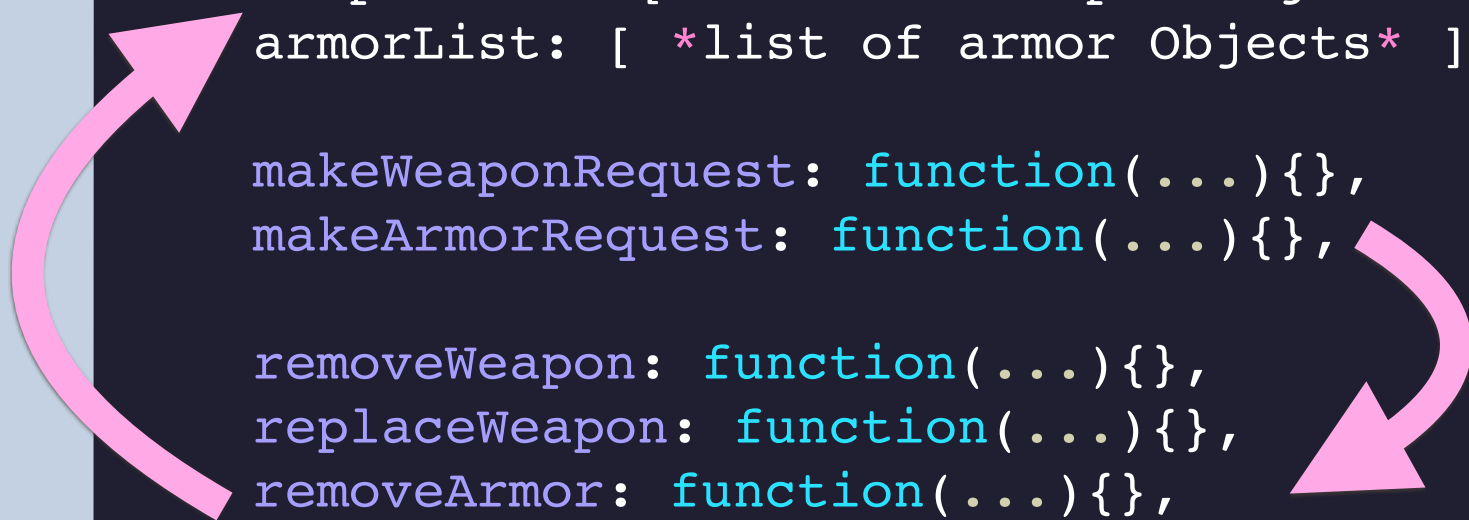


Making any modifications to the master lists should be a task only the module itself is allowed to undertake, so these remove/replace methods should be private.

PUBLIC METHODS AND VALUES OFTEN TRIGGER PRIVATE METHODS AND VALUES

In our case, public methods will signal the private methods to safely modify private data.

```
var ARMORY = {  
  
  weaponList: [ *list of weapon Objects* ],  
  armorList: [ *list of armor Objects* ],  
  
  makeWeaponRequest: function(...){},  
  makeArmorRequest: function(...){},  
  
  removeWeapon: function(...){},  
  replaceWeapon: function(...){},  
  removeArmor: function(...){},  
  replaceArmor: function(...){}  
  
};
```

A diagram with two pink curved arrows. One arrow starts from the 'makeWeaponRequest' and 'makeArmorRequest' methods and points to the 'weaponList' and 'armorList' arrays. The other arrow starts from the 'removeWeapon', 'replaceWeapon', 'removeArmor', and 'replaceArmor' methods and points to the same arrays, indicating that these methods interact with the private data.

Each request method will make some reference in its code to the remove/replace methods, which will have access to the weapons and armor lists. This makes closure very valuable.

JAVASCRIPT'S CLOSURE FEATURE WILL ALLOW US TO “PRIVATIZE” PROPERTIES

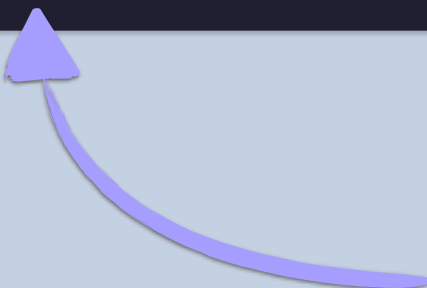
As a first visual step, we'll wrap the entire set of properties in an anonymous immediately invoked function expression (IIFE).

```
var ARMORY = {  
  
  weaponList: [ *list of weapon Objects* ],  
  armorList: [ *list of armor Objects* ],  
  
  makeWeaponRequest: function(...){},  
  makeArmorRequest: function(...){},  
  
  removeWeapon: function(...){},  
  replaceWeapon: function(...){},  
  removeArmor: function(...){},  
  replaceArmor: function(...){}  
  
};
```

JAVASCRIPT'S CLOSURE FEATURE WILL ALLOW US TO “PRIVATIZE” PROPERTIES

As a first visual step, we'll wrap the entire set of properties in an anonymous immediately invoked function expression (IIFE).

```
var ARMORY = (function(){  
  
    weaponList: [ *list of weapon Objects* ],  
    armorList: [ *list of armor Objects* ],  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    removeWeapon: function(...){},  
    replaceWeapon: function(...){},  
    removeArmor: function(...){},  
    replaceArmor: function(...){}  
  
})();
```



These last parentheses indicate that the function expression should be immediately executed.

NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

```
var ARMORY = (function(){  
  
    weaponList: [ *list of weapon Objects* ],  
    armorList: [ *list of armor Objects* ],  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    removeWeapon: function(...){},  
    replaceWeapon: function(...){},  
    removeArmor: function(...){},  
    replaceArmor: function(...){}  
  
})();
```

NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

```
var ARMORY = (function(){  
  
    weaponList  [ *list of weapon Objects* ]  
    armorList   [ *list of armor Objects* ]  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    removeWeapon  function(...){}  
    replaceWeapon function(...){}  
    removeArmor   function(...){}  
    replaceArmor  function(...){}  
  
})();
```

NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

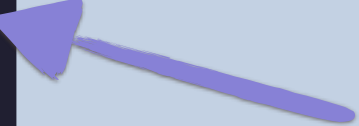
```
var ARMORY = (function(){  
  
    weaponList    [ *list of weapon Objects* ]  
    armorList     [ *list of armor Objects* ]  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    removeWeapon  function(...){}  
    replaceWeapon function(...){}  
    removeArmor   function(...){}  
    replaceArmor  function(...){}  
  
})();
```

NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

```
var ARMORY = (function(){  
  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    removeWeapon  function(...){}  
    replaceWeapon function(...){}  
    removeArmor   function(...){}  
    replaceArmor  function(...){}  
  
})();
```

The lists of data will become local variables for the immediately invoked function expression's scope. You'll see in a bit how this will make them private for the **ARMORY** namespace.



NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

```
var ARMORY = (function(){  
  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    removeWeapon  function(...){}  
    replaceWeapon function(...){}  
    removeArmor  function(...){}  
    replaceArmor  function(...){}  
  
})();
```


NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

```
var ARMORY = (function(){

    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];

    makeWeaponRequest: function(...){},
    makeArmorRequest: function(...){},

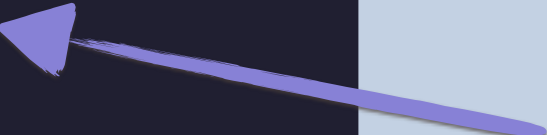
    removeWeapon    function(...){}
    replaceWeapon    function(...){}
    removeArmor     function(...){}
    replaceArmor     function(...){}

})();
```

NOW WE'LL MAKE DESIRED PRIVATE PROPERTIES INTO LOCAL EXECUTABLE CODE

These local values and methods will eventually be “closed up” into the armory’s namespace.

```
var ARMORY = (function(){  
  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){},  
  
    var removeWeapon = function(...){};  
    var replaceWeapon = function(...){};  
    var removeArmor = function(...){};  
    var replaceArmor = function(...){};  
  
})();
```



Same thing with these remove/replace methods, which will now belong to the function, instead of directly to the namespace. Stay with me...

NEXT, PULL EVERY PRIVATE VALUE AND METHOD TO THE TOP OF THE FUNCTION

We'll use good code organization and put all of the closure values near each other for easy reference.

```
var ARMORY = (function(){

    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];

    makeWeaponRequest: function(...){},
    makeArmorRequest: function(...){}

    var removeWeapon = function(...){};
    var replaceWeapon = function(...){};
    var removeArmor = function(...){};
    var replaceArmor = function(...){};

})();
```

NEXT, PULL EVERY PRIVATE VALUE AND METHOD TO THE TOP OF THE FUNCTION

We'll use good code organization and put all of the closure values near each other for easy reference.

```
var ARMORY = (function(){  
  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    var removeWeapon = function(...){};  
    var replaceWeapon = function(...){};  
    var removeArmor = function(...){};  
    var replaceArmor = function(...){};  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){}  
  
})();
```

HERE'S THE MONEY: TO MAKE SOME PROPERTIES PUBLIC, RETURN AN OBJECT.

Now we'll add the public properties to their own object, which will become the ARMORY namespace.

```
var ARMORY = (function(){

    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];

    var removeWeapon = function(...){};
    var replaceWeapon = function(...){};
    var removeArmor = function(...){};
    var replaceArmor = function(...){};

    makeWeaponRequest: function(...){},
    makeArmorRequest: function(...){}

})();
```


HERE'S THE MONEY: TO MAKE SOME PROPERTIES PUBLIC, RETURN AN OBJECT.

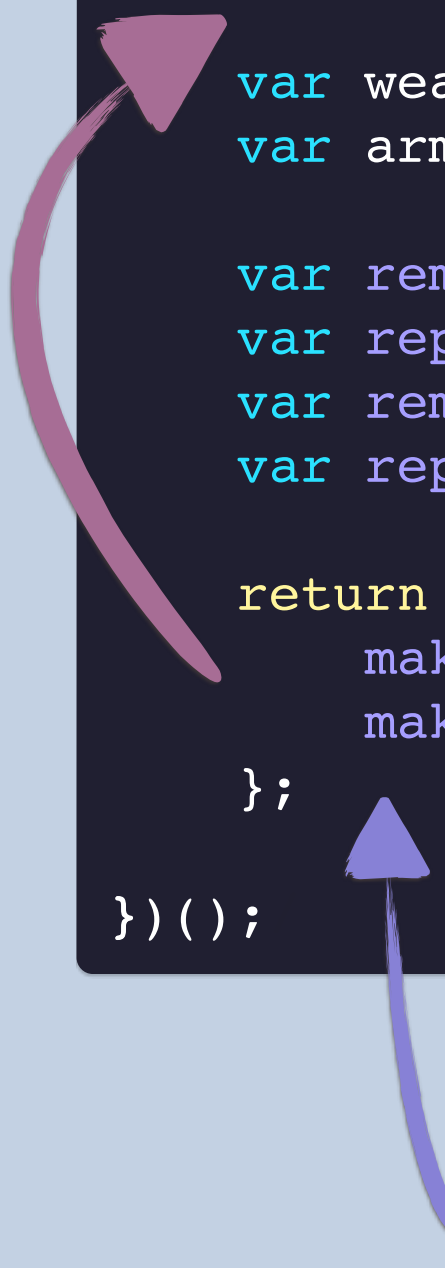
Now we'll add the public properties to their own object, which will become the ARMORY namespace.

```
var ARMORY = (function(){  
  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    var removeWeapon = function(...){};  
    var replaceWeapon = function(...){};  
    var removeArmor = function(...){};  
    var replaceArmor = function(...){};  
  
    makeWeaponRequest: function(...){},  
    makeArmorRequest: function(...){}  
  
})();
```

HERE'S THE MONEY: TO MAKE SOME PROPERTIES PUBLIC, RETURN AN OBJECT.

Now we'll add the public properties to their own object, which will become the ARMORY namespace.

```
var ARMORY = (function(){  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    var removeWeapon = function(...){};  
    var replaceWeapon = function(...){};  
    var removeArmor = function(...){};  
    var replaceArmor = function(...){};  
  
    return {  
        makeWeaponRequest: function(...){},  
        makeArmorRequest: function(...){}  
    };  
})();
```




Because the function expression is actually called, this returned object will be handed immediately to the **ARMORY** variable and become a namespace.

CLOSURE NOW PRODUCES OUR DESIRED PRIVATE METHODS AND VALUES

All of the function's local variables are “bound down” within the scope of the returned namespace object.

```
var ARMORY = (function(){  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    var removeWeapon = function(...){};  
    var replaceWeapon = function(...){};  
    var removeArmor = function(...){};  
    var replaceArmor = function(...){};  
  
    return {  
        makeWeaponRequest: function(...){},  
        makeArmorRequest: function(...){}  
    };  
})();
```



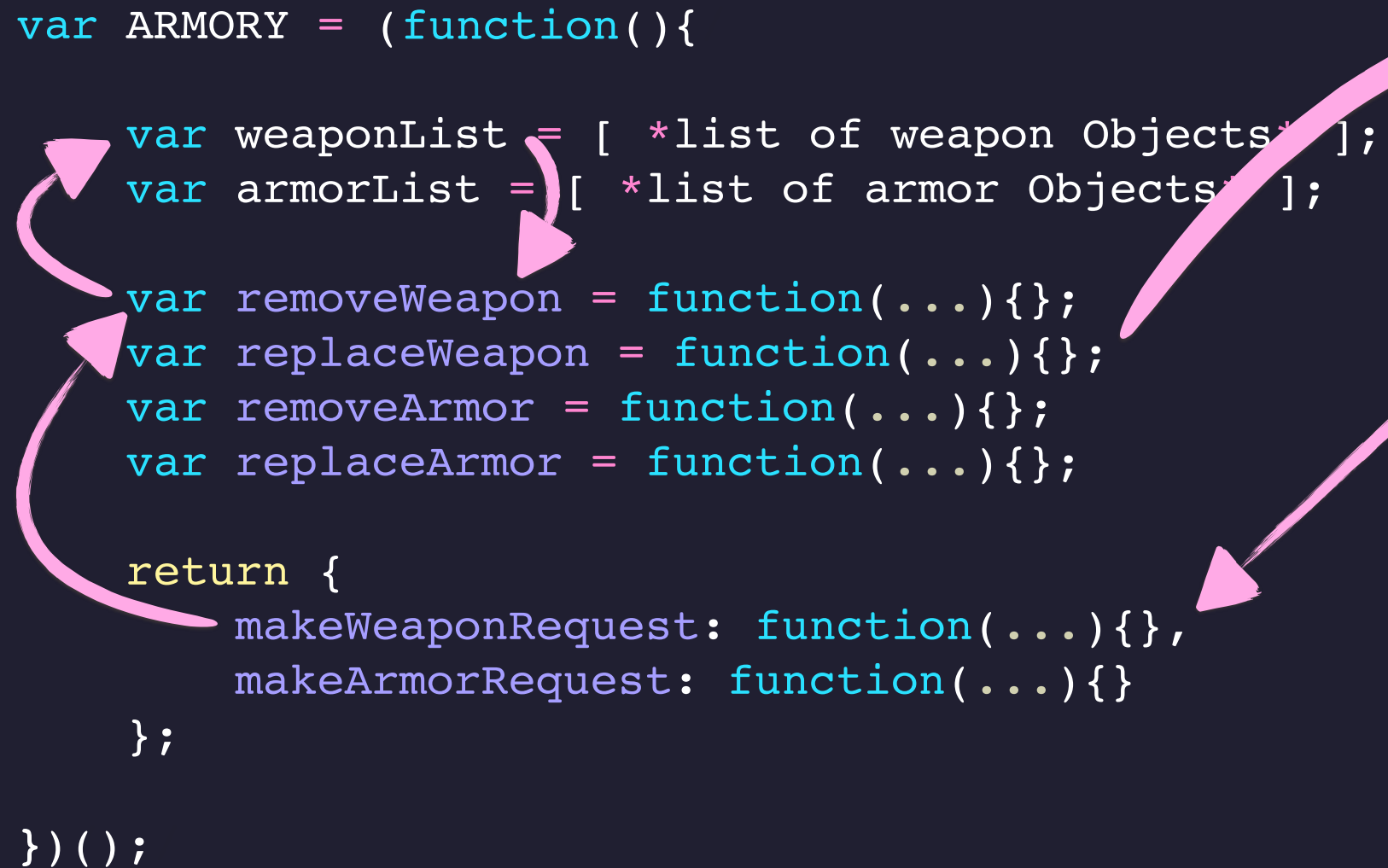
Notice that none of our function's local variables are ever properties within the returned namespace object...

...but they are there nonetheless, visible to and able to be referenced by ONLY the members of the local namespace scope.

THE BASICS OF OUR MODULE ARE NOW COMPLETE

Our sensitive data is private by closure, and our public properties are accessible through the namespace.

```
var ARMORY = (function(){  
  
    var weaponList = [ *list of weapon Objects* ];  
    var armorList = [ *list of armor Objects* ];  
  
    var removeWeapon = function(...){};  
    var replaceWeapon = function(...){};  
    var removeArmor = function(...){};  
    var replaceArmor = function(...){};  
  
    return {  
        makeWeaponRequest: function(...){},  
        makeArmorRequest: function(...){}  
    };  
})();
```



```
ARMORY.makeWeaponRequest("Excalibur");
```



Calls an invisible `removeWeapon` function to try to get Excalibur.

If some conditions are met, the invisible `removeWeapon` method deletes and retrieves an object from an invisible `weaponList`.

The `removeWeapon` returns the object for use to the scope of `makeWeaponRequest`.

THE MAIL OF MODULARITY

Section 2

Anonymous Closures

JAVASCRIPT
BEST PRACTICES

THE MAIL OF MODULARITY

Section 3
Global Imports

JAVASCRIPT
BEST PRACTICES

OUR CURRENT MODULE SEEMS TO ONLY NEED LOCAL VARIABLES

If we look a little closer, we might find that a global variable is involved in one of our methods.

```
var ARMORY = (function(){

    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];

    var removeWeapon = function(...){};
    var replaceWeapon = function(...){};
    var removeArmor = function(...){};
    var replaceArmor = function(...){};

    return {
        makeWeaponRequest: function(...){},
        makeArmorRequest: function(...){}
    };

})();
```

OUR CURRENT MODULE SEEMS TO ONLY NEED LOCAL VARIABLES

If we look a little closer, we might find that a global variable is involved in one of our methods.

```
var wartime = true;
var ARMORY = (function(){

    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];

    var removeWeapon = function(...){};
    var replaceWeapon = function(...){};
    var removeArmor = function(...){};
    var replaceArmor = function(...){};

    return {
        makeWeaponRequest: function(...){},
        makeArmorRequest: function(...){}
    };

})();
```

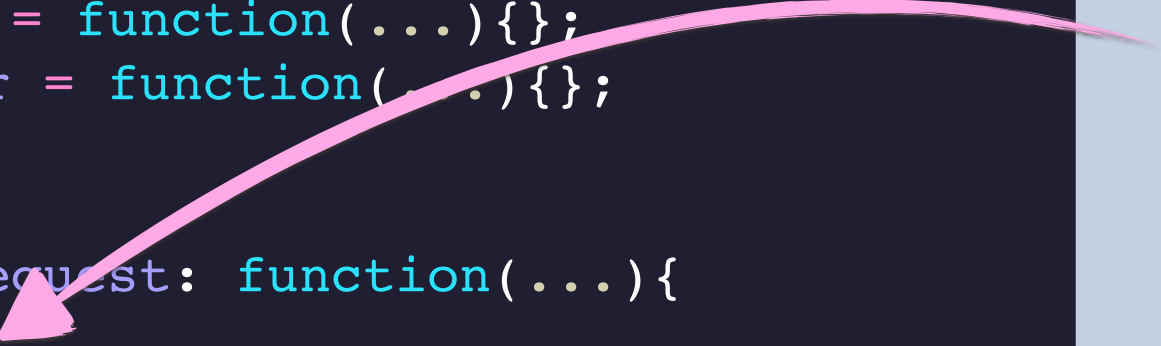
Somewhere in our global scope, there's a variable that signals whether the kingdom is currently at war.

OUR CURRENT MODULE SEEMS TO ONLY NEED LOCAL VARIABLES

If we look a little closer, we might find that a global variable is involved in one of our methods.

```
var wartime = true;
var ARMORY = (function(){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(wartime) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```



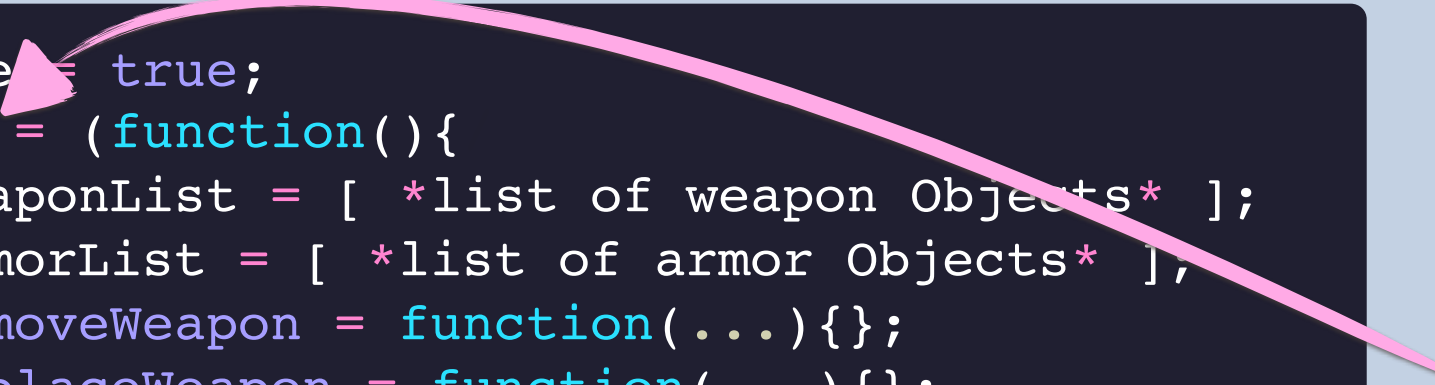
Our `makeWeaponRequest` function checks to see if war exists; if so, we'll let both knights and civilians have weaponry.

STANDARD GLOBAL USE IN A MODULE PATTERN CAUSES TWO PROBLEMS

#1: When non-local variables are referenced in a module, the entire length of the scope chain is checked.

```
var wartime = true;
var ARMORY = (function(){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ],
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(wartime) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```



First, the local scope of the namespace would be checked.

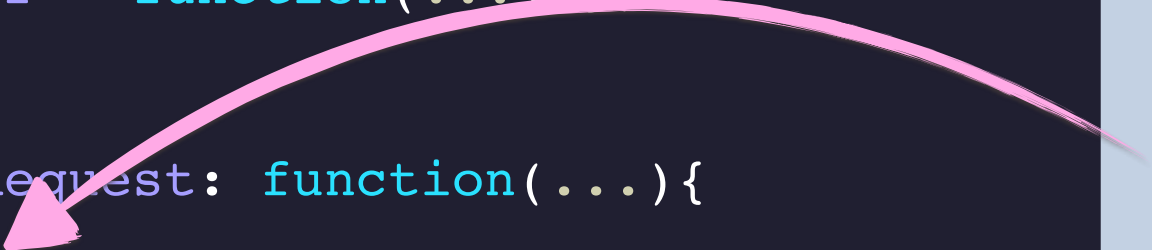
Then, if the namespace happened to be nested, the outer namespace would also be checked...and so on toward the global scope.

STANDARD GLOBAL USE IN A MODULE PATTERN CAUSES TWO PROBLEMS

#1: When non-local variables are referenced in a module, the entire length of the scope chain is checked.

```
var wartime = true;
var ARMORY = (function(){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){}.

  return {
    makeWeaponRequest: function(...){
      if(wartime) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```



This happens every time **wartime** is encountered throughout the module...a very expensive process if there is any namespace depth and/or multiple references.

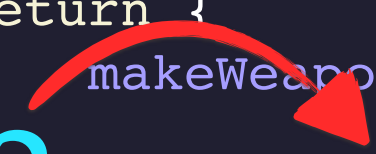
STANDARD GLOBAL USE IN A MODULE PATTERN CAUSES TWO PROBLEMS

#2: Lengthy namespaces mean that global variables have unclear scope, leading to code that is tough to manage.

```
var wartime = true;
var ARMORY = (function(){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(wartime) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```

???



Developers that encounter externally scoped variables may be unable to immediately place the source of the variable's data.

They may have to look through a lot of code to figure out where the data came from, or they may make possibly dangerous changes to values if there's an incorrect local assumption.

FOR CLEARER, FASTER GLOBALS IN MODULES, USE IMPORTS

While devs may still need to review code, importing globals creates faster local variables that are (more) clear in source.

```
var wartime = true;
var ARMORY = (function(){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};


  return {
    makeWeaponRequest: function(...){
      if(wartime) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```

FOR CLEARER, FASTER GLOBALS IN MODULES, USE IMPORTS

While devs may still need to review code, importing globals creates faster local variables that are (more) clear in source.

```
var wartime = true;
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(wartime) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```



The first step is to create a parameter for the immediately invoked function expression that returns our namespace object. You can create as many parameters as there are globals that you'll be importing.

FOR CLEARER, FASTER GLOBALS IN MODULES, USE IMPORTS

While devs may still need to review code, importing globals creates faster local variables that are (more) clear in source.

```
var wartime = true;
var ARMORY = (function( war ){
    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];
    var removeWeapon = function(...){};
    var replaceWeapon = function(...){};
    var removeArmor = function(...){};
    var replaceArmor = function(...){};

    return {
        makeWeaponRequest: function(...){
            if(          ) //let civilians have weaponry
        },
        makeArmorRequest: function(...){}
    };
})();
```

FOR CLEARER, FASTER GLOBALS IN MODULES, USE IMPORTS

While devs may still need to review code, importing globals creates faster local variables that are (more) clear in source.

```
var wartime = true;
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if( war ) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```

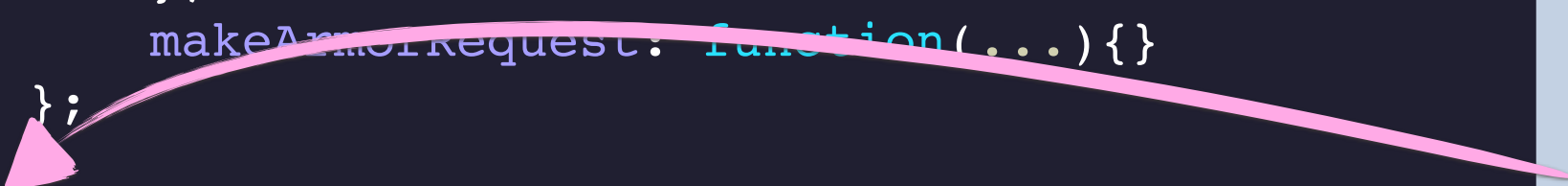
Next, replace global names with parameter names.

FOR CLEARER, FASTER GLOBALS IN MODULES, USE IMPORTS

While devs may still need to review code, importing globals creates faster local variables that are (more) clear in source.

```
var wartime = true;
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})();
```



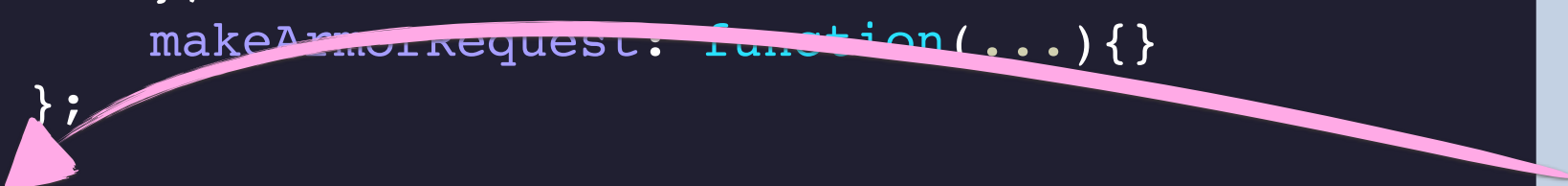
Lastly, pass all your globals into your IIFE using the calling parentheses. Yeah, imports!

FOR CLEARER, FASTER GLOBALS IN MODULES, USE IMPORTS

While devs may still need to review code, importing globals creates faster local variables that are (more) clear in source.

```
var wartime = true;
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```



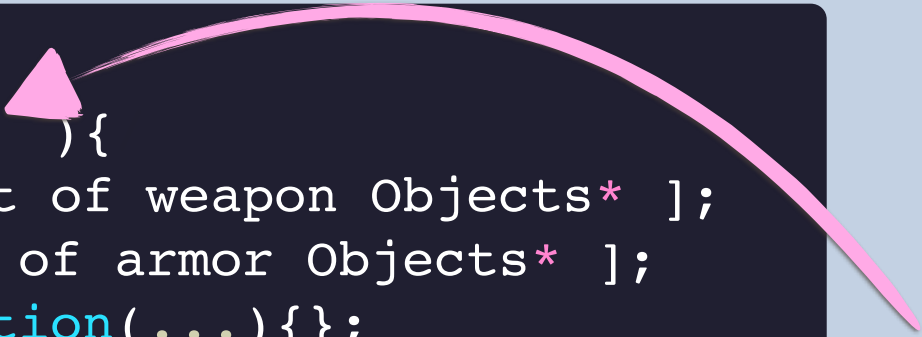
Lastly, pass all your globals into your IIFE using the calling parentheses. Yeah, imports!

IMPORTS ARE CLOSED UP AS LOCAL VARIABLES

An imported global variable becomes another piece of data, boxed up in the module's closure.

```
var wartime = true;
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```



Bonus! Now the function's parameter creates a modifiable value for use in the module, while the global value stays protected if necessary.

THE MAIL OF MODULARITY

Section 3
Global Imports

JAVASCRIPT
BEST PRACTICES

THE MAIL OF MODULARITY

Section 4
Augmentation

JAVASCRIPT
BEST PRACTICES

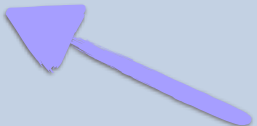
MODULES OFTEN NEED TO HAVE ADDITIONS TO THEIR EXISTING PROPERTIES

If you've ever worked in a large base of code documents, you know the value of splitting some functionality between files.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```



Here's the file where the armory module is built.

MODULES OFTEN NEED TO HAVE ADDITIONS TO THEIR EXISTING PROPERTIES

If you've ever worked in a large base of code documents, you know the value of splitting some functionality between files.

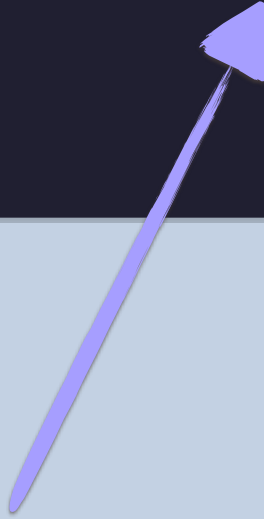
ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```



Here's the file where global information for the kingdom is declared.

AUGMENTATION PROVIDES EXTRA PROPERTIES FOR EXISTING MODULES

In a separate file we'll keep a function which will add values or functionality to our existing Armory.

ARMORY.JS


```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS



This file will add functionality to the Armory in the event of war.

AUGMENTATION PROVIDES EXTRA PROPERTIES FOR EXISTING MODULES

In a separate file we'll keep a function which will add values or functionality to our existing Armory.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

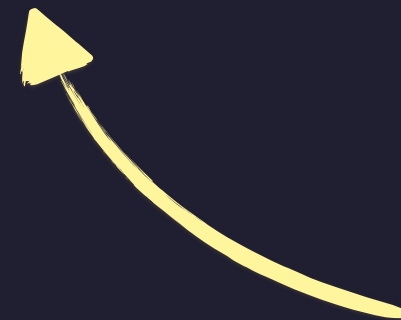
  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function(      ){
```



We'll assign an updated object to the existing global **ARMORY** namespace.

```
)(      );
```

AUGMENTATION PROVIDES EXTRA PROPERTIES FOR EXISTING MODULES

In a separate file we'll keep a function which will add values or functionality to our existing Armory.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };

})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function( oldNS ){
```



Since our namespace is global, we'll import it as a local, in order to make some modifications to a temporary object.

```
)(
  );
```

AUGMENTATION PROVIDES EXTRA PROPERTIES FOR EXISTING MODULES

In a separate file we'll keep a function which will add values or functionality to our existing Armory.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function( oldNS ){
```

We pass in the old module to our modifying IIFE, and the result will get assigned to the place where the old module was!



```
)( ARMORY );
```

AUGMENTATION PROVIDES EXTRA PROPERTIES FOR EXISTING MODULES

In a separate file we'll keep a function which will add values or functionality to our existing Armory.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

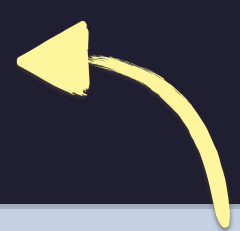
  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function( oldNS ){
  var oilBarrels = 1000;
  var catapults = ["Stoneslinger",
                  "Rockrain",
                  "The Giant's Arm" ];
  oldNS.assignCatapult = function (regiment){
    //hooks up a regiment with a sweet catapult
    //and some oil barrels!
  };
  return oldNS;
})( ARMORY );
```



We add all some new private values and public functionality that we desire, and then return the modified module!

BEWARE: PREVIOUS PRIVATE DATA WILL NOT BE ACCESSIBLE TO THE NEW PROPERTIES

Augmented properties will have access a new private state, if used, but not to the other file's closed data.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

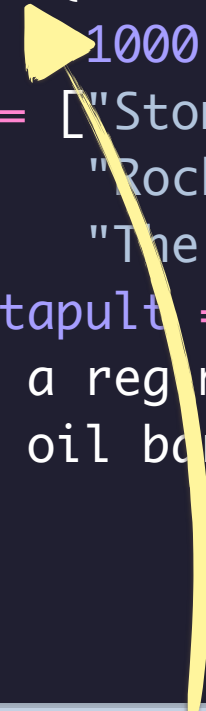
  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function( oldNS ){
  var oilBarrels = 1000;
  var catapults = ["Stoneslinger",
                  "Rockrain",
                  "The Giant's Arm" ];
  oldNS.assignCatapult = function (regiment){
    //hooks up a regiment with a sweet catapult
    //and some oil barrels!
  };
  return oldNS;
})( ARMORY );
```



Remember that closures are produced by the function itself, not the returned Object. A new function expression won't recreate those old references.

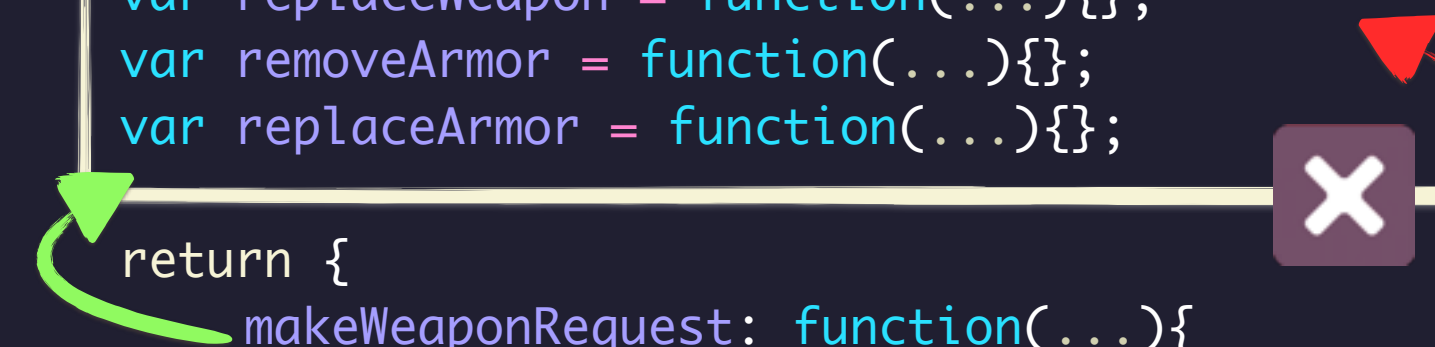
BEWARE: PREVIOUS PRIVATE DATA WILL NOT BE ACCESSIBLE TO THE NEW PROPERTIES

Augmented properties will have access a new private state, if used, but not to the other file's closed data.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```




GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function( oldNS ){
  var oilBarrels = 1000;
  var catapults = ["Stoneslinger",
    "Rockrain",
    "The Giant's Arm" ];
  oldNS.assignCatapult = function (regiment){
    //hooks up a regiment with a sweet catapult
    //and some oil barrels!
  };
  return oldNS;
})( ARMORY );
```



Any new properties will have no access to the private data from the earlier closure. The earlier public properties, however, will retain the reference.

BEST PRACTICE: GROUP FILE CONTENTS AROUND NEEDED DATA

Any cross-file private state build won't have the level of privacy of a single closure, and often leads to hard-to-manage code.

ARMORY.JS

```
var ARMORY = (function( war ){
  var weaponList = [ *list of weapon Objects* ];
  var armorList = [ *list of armor Objects* ];
  var removeWeapon = function(...){};
  var replaceWeapon = function(...){};
  var removeArmor = function(...){};
  var replaceArmor = function(...){};

  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(...){}
  };
})(wartime);
```

GLOBALKINGDOM.JS

```
...
var wartime = true;
...
```

WARCHEST.JS

```
ARMORY = (function( oldNS ){
  var oilBarrels = 1000;
  var catapults = ["Stoneslinger",
                  "Rockrain",
                  "The Giant's Arm" ];
  oldNS.assignCatapult = function (regiment){
    //hooks up a regiment with a sweet catapult
    //and some oil barrels!
  };
  return oldNS;
})( ARMORY );
```



Our assignCatapults method wouldn't need access to the private data in the original module, so this augmentation will not hold any broken references.

THE MAIL OF MODULARITY

Section 4
Augmentation

JAVASCRIPT
BEST PRACTICES