

THE SWORD OF SYNTAX

Section 1

Ternary Conditionals

JAVASCRIPT
BEST PRACTICES

OUR STANDARD CONDITIONAL

Some simple conditionals take a lot of code to choose an assignment.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



This takes two blocks of code just to get one assignment finished. Let's use a syntax trick to make this more concise!



A NEW KIND OF CONDITIONAL

The ternary conditional provides a shortcut over lengthier conditional blocks.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



someCondition



To build a ternary conditional, the condition to check is placed down first...

A NEW KIND OF CONDITIONAL

The ternary conditional provides a shortcut over lengthier conditional blocks.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



```
someCondition ?
```



... and that's followed by the question mark operator, which divides the conditional into “condition” and “actions to take”.

A NEW KIND OF CONDITIONAL

The ternary conditional provides a shortcut over lengthier conditional blocks.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



```
someCondition ? pickThisIfTrue : pickThisIfFalse;
```



Next, a pair of expressions separated by a colon. One of these will be picked as the conditional's result!

A NEW KIND OF CONDITIONAL

The ternary conditional provides a shortcut over lengthier conditional blocks.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



```
someCondition ? pickThisIfTrue : pickThisIfFalse;
```



If the condition's true,
we'll get the left side as
a result.

A NEW KIND OF CONDITIONAL

The ternary conditional provides a shortcut over lengthier conditional blocks.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



```
someCondition ? pickThisIfTrue : pickThisIfFalse;
```



If the condition's false, we
get the right side.

BUILDING OUR OWN TERNARY

Let's build our weapon-choosing conditional block as a ternary conditional.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

Since `isArthur` is currently false, we get the right side returned from this ternary conditional.

```
someCondition ? pickThisIfTrue : pickThisIfFalse;
```

```
isArthur ? "Excalibur" : "Longsword";
```

→ "Longsword"



BUILDING OUR OWN TERNARY

Let's build our weapon-choosing conditional block as a ternary conditional.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

By itself, this ternary conditional expression returns the right answer, but what happened to storing it in `weapon`?

```
someCondition ? pickThisIfTrue : pickThisIfFalse;
```

```
isArthur ? "Excalibur" : "Longsword";
```

→ "Longsword"



ASSIGNING RESULTS OF TERNARIES

Since the ternary conditional always returns a result, we can assign it!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



```
someCondition ? pickThisIfTrue : pickThisIfFalse;
```

```
var weapon = isArthur ? "Excalibur" : "Longsword";
```



→ "Longsword"

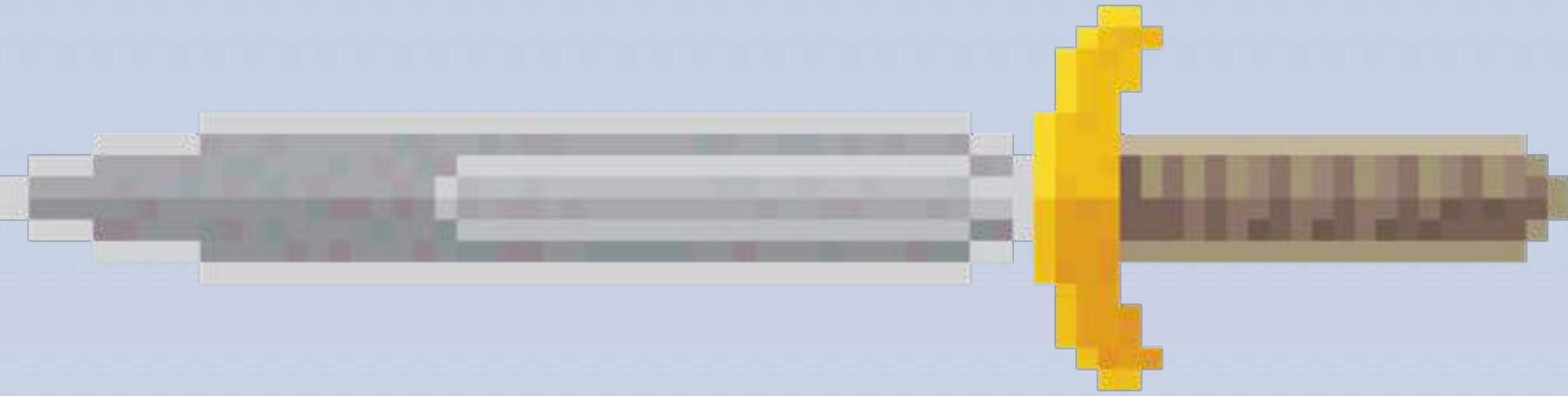
After choosing the appropriate sword, the compiler now assigns that choice to `weapon`.

ASSIGNING RESULTS OF TERNARIES

Since the ternary conditional always returns a result, we can assign it!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```



```
someCondition ? pickThisIfTrue : pickThisIfFalse;  
  
var weapon = isArthur ? "Excalibur" : "Longsword";  
console.log("Current weapon: " + weapon);
```

→ Current weapon: Longsword

ASSIGNING RESULTS OF TERNARIES

Since the ternary conditional always returns a result, we can assign it!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
var weapon = isArthur ? "Excalibur" : "Longsword";  
console.log("Current weapon: " + weapon);
```



USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
var weapon = isArthur ? "Excalibur" : "Longsword";  
console.log("Current weapon: " + weapon);
```



We can use the entire ternary as
an expression in concatenation.



USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
isArthur ? "Excalibur" : "Longsword"  
console.log("Current weapon: " + );
```



USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
isArthur ? "Excalibur" : "Longsword"  
console.log("Current weapon: " + );
```



USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
isArthur ? "Excalibur" : "Longsword"  
console.log("Current weapon: " + );
```



USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



 → Excalibur

Uh oh.
What happened?

USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



→ Excalibur

The `?` has a lower precedence than
the `+` concatenation operator.

USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



→ Excalibur

USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

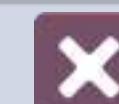
```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



The `+` only knows to evaluate a variable and add it to a string, all before the `?` gets to check a condition.



→ Excalibur

"Current weapon: "  isArthur → "Current weapon: false"
String Boolean String

USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;  
  
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}  
  
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



X → Excalibur

"Current weapon: false"
String

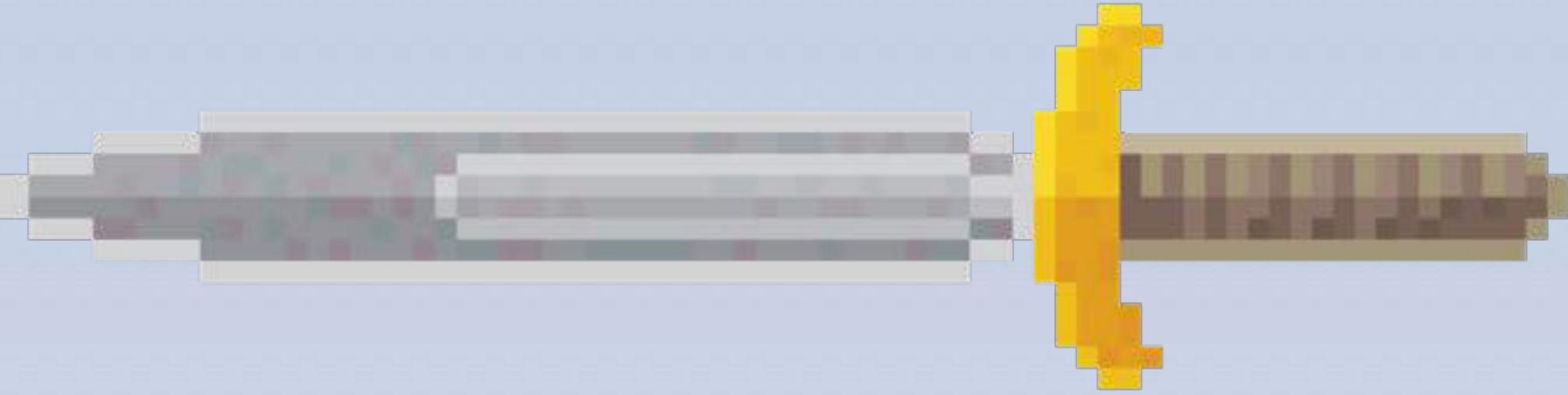
USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



→ Excalibur

"Current weapon: false"

String

USING TERNARIES AS EXPRESSIONS

If we only need the result once, we can even eliminate the assignment!

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



The `?` now looks for a boolean...but finds a string. Turns out, any JS value that is not `false`, `0`, `undefined`, `Nan`, `""`, or `null` will always evaluate as “truthy”.



→ Excalibur

"Current weapon: false"

String

?

"Excalibur" : "Longsword"

ENSURE TERNARIES ARE ISOLATED

Use parentheses to ensure the conditional is checked correctly.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + isArthur ? "Excalibur" : "Longsword");
```



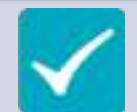
ENSURE TERNARIES ARE ISOLATED

Use parentheses to ensure the conditional is checked correctly.

```
var isArthur = false;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + (isArthur ? "Excalibur" : "Longsword"));
```



→ Current weapon: Longsword

ENSURE TERNARIES ARE ISOLATED

Use parentheses to ensure the conditional is checked correctly.

```
var isArthur = true;  
var weapon;
```

```
if(isArthur){  
    weapon = "Excalibur";  
} else {  
    weapon = "Longsword";  
}
```

```
console.log("Current weapon: " + (isArthur ? "Excalibur" : "Longsword"));
```



→ Current weapon: Excalibur

COMPOUND TERNARY CONDITIONS

We can use compound Boolean expressions to make ternary decisions, too.

```
var isArthur = true;
```



```
console.log("Current weapon: " + (isArthur ? "Excalibur" : "Longsword"));
```

COMPOUND TERNARY CONDITIONS

We can use compound Boolean expressions to make ternary decisions, too.

```
var isArthur = true;  
var isKing = false;
```



```
console.log("Current weapon: " + (isArthur && isKing ? "Excalibur" : "Longsword"));
```



→ Current weapon: Longsword

COMPOUND TERNARY CONDITIONS

We can use compound Boolean expressions to make ternary decisions, too.

```
var isArthur = true;  
var isKing = true;
```



```
console.log("Current weapon: " + (isArthur && isKing ? "Excalibur" : "Longsword"));
```



→ Current weapon: Excalibur

TERNARIES CAN TAKE ACTION

Any executable statement can serve as a ternary's response choices.

```
var isArthur = true;  
var isKing = true;
```



```
console.log("Current weapon: " + (isArthur && isKing ? "Excalibur" : "Longsword"));
```



→ Current weapon: Excalibur

TERNARIES CAN TAKE ACTION

Any executable statement can serve as a ternary's response choices.

```
var isArthur = true;  
var isKing = true;
```

```
isArthur && isKing ? : 
```

TERNARIES CAN TAKE ACTION

Any executable statement can serve as a ternary's response choices.

```
var isArthur = true;  
var isKing = true;
```

```
isArthur && isKing ? alert("Hail Arthur, King of the Britons!") : 
```

TERNARIES CAN TAKE ACTION

Any executable statement can serve as a ternary's response choices.

```
var isArthur = true;  
var isKing = true;
```



```
isArthur && isKing ? alert("Hail Arthur, King of the Britons!") :  
alert("Charge on, ye Knight, for the glory of the King!");
```

TERNARIES CAN TAKE ACTION

Any executable statement can serve as a ternary's response choices.

```
var isArthur = true;  
var isKing = false;
```



The page at <https://www.codeschool.com>
says:

Charge on, ye Knight, for the glory of the King!

OK

```
isArthur && isKing ? alert("Hail Arthur, King of the Britons!") :  
alert("Charge on, ye Knight, for the glory of the King!");
```

BUILD AND CHOOSE FUNCTIONS ON THE FLY

Ternaries provide a different format for picking immediately-invoked functions.

```
var isArthur = true;  
var isKing = false;
```

```
isArthur && isKing ? : 
```

BUILD AND CHOOSE FUNCTIONS ON THE FLY

Ternaries provide a different format for picking immediately-invoked functions.

```
var isArthur = true;  
var isKing = false;
```



```
isArthur && isKing ? function (){  
    alert("Hail Arthur, King of the Britons!");  
    console.log("Current weapon: Excalibur");  
}()  
:  
function (){  
    alert("Charge on, ye Knight, for the glory of the King!");  
    console.log("Current weapon: Longsword");  
}();
```

Remember that adding the parentheses calls the function expression.

A purple arrow points from the word "expression" in the text above to the closing parenthesis of the second function definition in the code.

→ Current weapon: Longsword

MULTIPLE ACTIONS IN TERNARIES

Each result option provides the opportunity to execute multiple actions.

```
var isArthur = true;  
var isKing = false;
```

```
isArthur && isKing ?
```

```
:
```

MULTIPLE ACTIONS IN TERNARIES

Each result option provides the opportunity to execute multiple actions.

```
var isArthur = true;  
var isKing = false;  
var weapon;  
var helmet;
```

```
isArthur && isKing ? (weapon = "Excalibur", helmet = "Goosewhite")  
:  
(weapon = "Longsword", helmet = "Iron Helm");
```



Multiple statements within a single ternary response are grouped in parentheses and separated by a comma.

MULTIPLE ACTIONS IN TERNARIES

Each result option provides the opportunity to execute multiple actions.

```
var isArthur = true;  
var isKing = false;  
var weapon;  
var helmet;
```

```
isArthur && isKing ? (weapon = "Excalibur", helmet = "Goosewhite")  
:  
(weapon = "Longsword", helmet = "Iron Helm");
```

```
console.log("Current weapon: " + weapon + "\nCurrent helmet: " + helmet);
```

→ Current weapon: Longsword
Current helmet: Iron Helm

LASTLY, TERNARIES CAN BE NESTED!

A ternary can hold other ternaries within each of the possible responses.

```
var isArthur = true;  
var isKing = false;  
var weapon;  
var helmet;
```

```
isArthur && isKing ? ( weapon = "Excalibur", helmet = "Goosewhite")  
:  
( weapon = "Longsword", helmet = "Iron Helm");
```

LASTLY, TERNARIES CAN BE NESTED!

A ternary can hold other ternaries within each of the possible responses.

```
var isArthur = true;  
var isKing = false;  
var isArcher = true;  
var weapon;  
var helmet;
```

A nested, multi-action ternary!

```
isArthur && isKing ? (weapon = "Excalibur", helmet = "Goosewhite")  
:  
isArcher ? (weapon = "Longbow", helmet = "Mail Helm")  
: (weapon = "Longsword", helmet = "Iron Helm");
```

```
console.log("Current weapon: " + weapon + "\nCurrent helmet: " + helmet);
```

→ Current weapon: Longbow
Current helmet: Mail Helm

THE SWORD OF SYNTAX

Section 2
Logical Assignment I:
The “OR” Operator

JAVASCRIPT
BEST PRACTICES

ASSIGNMENTS WITH LOGICAL OPERATORS

Logical operators can make conditional assignments even shorter than ternaries.

```
var armory = { addSword: function (sword) {  
    }  
};
```



Our armory object will initially contain a function property for adding swords.

ASSIGNMENTS WITH LOGICAL OPERATORS

Logical operators can make conditional assignments even shorter than ternaries.

```
var armory = { addSword: function (sword) {  
    this.swords = this.swords ? this.swords : [ ];  
}  
};
```



If the `swords` property currently exists (i.e., it isn't "falsy"), then the list it contains will just stay the same.

ASSIGNMENTS WITH LOGICAL OPERATORS

Logical operators can make conditional assignments even shorter than ternaries.

```
var armory = { addSword: function (sword) {  
    this.swords = this.swords ? this.swords : [ ];  
}  
};
```



But if it doesn't exist, it will be created as an empty list, and then assigned to a newly created **swords** property.

ASSIGNMENTS WITH LOGICAL OPERATORS

Logical operators can make conditional assignments even shorter than ternaries.

```
var armory = { addSword: function (sword) {  
    this.swords = this.swords ? this.swords : [ ];  
    this.swords.push(sword);  
}  
};
```



Then, with whatever list we've now got in the `swords` property, we add the new `sword` to our `armory`.

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {
    this.swords = this.swords ? this.swords : [ ];
    this.swords.push(sword);
}
};
```

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {  
    this.swords = [ ];  
    this.swords.push(sword);  
}  
};
```



The question mark, the condition to check, and the colon disappear ...

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {
    this.swords = this.swords || [];
    this.swords.push(sword);
}
};
```

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {  
    this.swords = this.swords || [ ];  
    this.swords.push(sword);  
}  
};
```



When used in assignment, the OR operator will try to select the first value it encounters that is not “falsy.”

If Swords is empty ...

```
this.swords = this.swords || [ ];  
undefined || [ ];  
  
this.swords = [ ];
```

If Swords has contents ...

```
this.swords = this.swords || [ ];  
[ ... ] || [ ];  
  
this.swords = [ ... ];
```

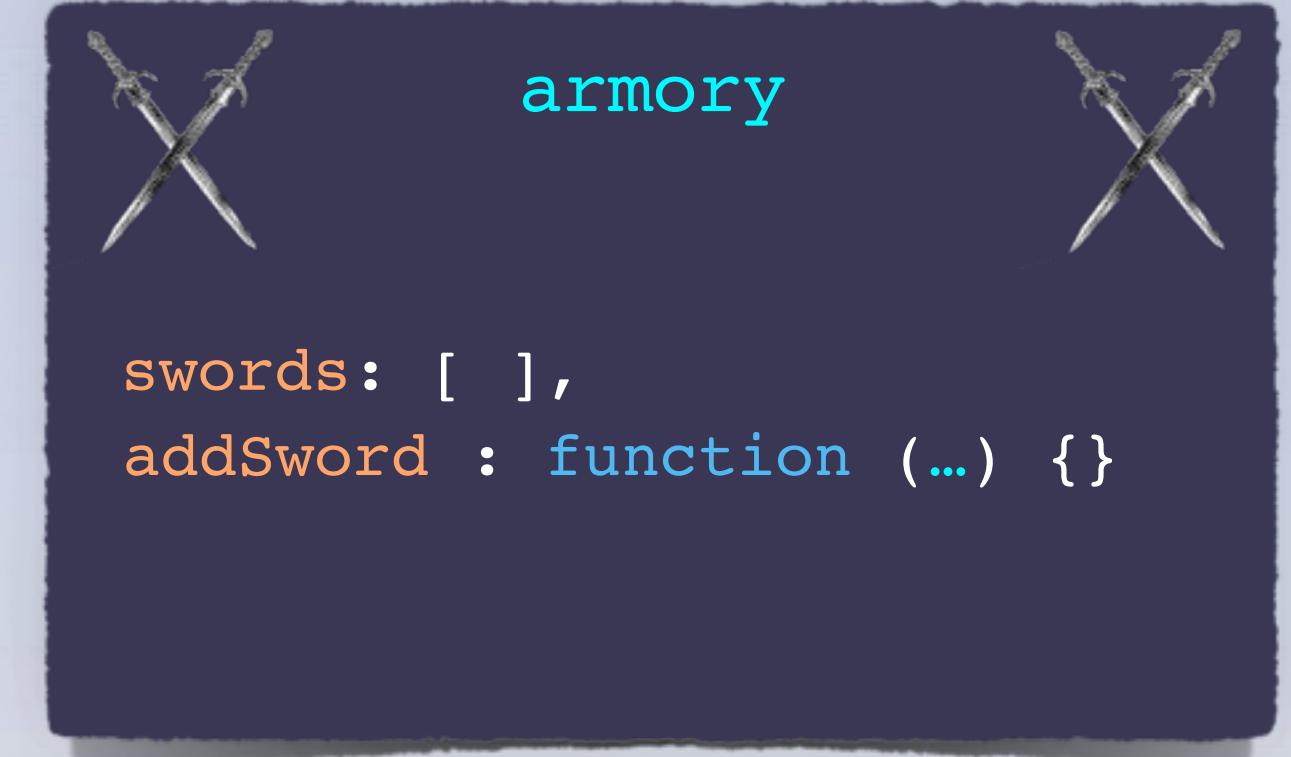
As soon as something “not false” is found, it is accepted for the assignment. In this case, our [] never even gets looked at. This event is often called “short-circuiting.”

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {  
    this.swords = this.swords | [ ];  
    this.swords.push(sword);  
}  
};
```

```
armory.addSword("Broadsword");
```



ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = {   addSword: function (sword) {  
    this.swords = this.swords || [ ];  
    this.swords.push(sword);  
  }  
};
```

```
armory.addSword("Broadsword");  
armory.addSword("Katana");
```



armory



```
swords: ["Broadsword"],  
addSword : function (...) {}
```

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = {   addSword: function (sword) {  
    this.swords = this.swords || [ ];  
    this.swords.push(sword);  
  }  
};
```

```
armory.addSword("Broadsword");  
armory.addSword("Katana");  
armory.addSword("Claymore");
```



armory

```
swords: [ "Broadsword"  
          "Katana" ],
```



```
addSword : function (...) {}
```

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = {   addSword: function (sword) {  
    this.swords = this.swords || [ ];  
    this.swords.push(sword);  
  }  
};
```

```
armory.addSword("Broadsword");  
armory.addSword("Katana");  
armory.addSword("Claymore");
```



armory

```
swords: [ "Broadsword"  
          "Katana" ],
```



```
addSword : function (...) {}
```

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {
    this.swords = this.swords || [ ];
    this.swords.push(sword);
}
};
```

```
armory.addSword("Broadsword");
armory.addSword("Katana");
armory.addSword("Claymore");
armory.addSword("Scimitar");
```



```
armory
swords: [ "Broadsword"
           "Katana"
           "Claymore"] ,
addSword : function (...) {}
```

A dark blue rectangular card with two crossed swords icons at the top. The word "armory" is written in cyan. Below it, the "swords" property is shown as an array of three swords: "Broadsword", "Katana", and "Claymore". The "addSword" method is also listed.

ASSIGNMENTS WITH LOGICAL OPERATORS

Using a logical assignment here will eliminate a read of the sword property.

```
var armory = { addSword: function (sword) {
    this.swords = this.swords || [ ];
    this.swords.push(sword);
}
};
```

```
armory.addSword("Broadsword");
armory.addSword("Katana");
armory.addSword("Claymore");
armory.addSword("Scimitar");
```



```
console.log(armory.swords);
→ ["Broadsword", "Katana", "Claymore", "Scimitar"]
```



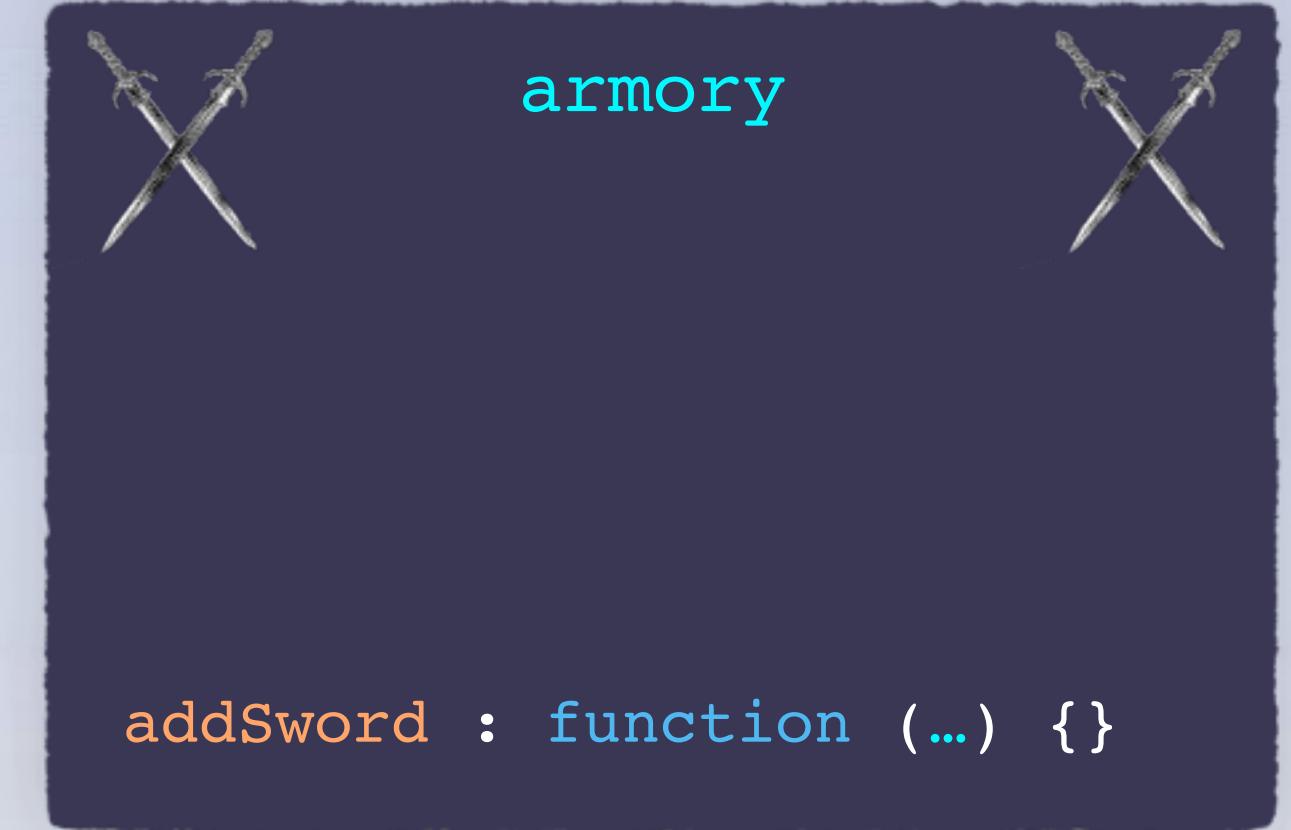
```
armory
swords: [ "Broadsword"
           "Katana"
           "Claymore"
           "Scimitar" ],
addSword : function (...) { }
```



A CLOSER LOOK AT THAT “OR” ASSIGNMENT

The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

```
var armory = { addSword: function (sword) {
    this.swords = this.swords || [ ];
    this.swords.push(sword);
}
};
```



A CLOSER LOOK AT THAT “OR” ASSIGNMENT

The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

```
var armory = { addSword: function (sword) {  
    this.swords = [ ] || this.swords;  
    this.swords.push(sword);  
}  
};
```

When used in assignment, the OR operator will try to select the first value it encounters that is not “falsy.”

```
armory.addSword("Broadsword");
```

The diagram illustrates the state of the `armory` object. It consists of three parts: a top section with crossed swords, a middle section labeled `armory`, and a bottom section with crossed swords. The middle section contains the following code:

```
swords: [ ],  
addSword : function (...) {}
```

A CLOSER LOOK AT THAT “OR” ASSIGNMENT

The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

```
var armory = { addSword: function (sword) {  
    this.swords = [ ] || this.swords;  
    this.swords.push(sword);  
}  
};
```

When used in assignment, the OR operator will try to select the first value it encounters that is not “falsy.”

```
armory.addSword("Broadsword");  
armory.addSword("Katana");
```



armory
swords: ["Broadsword"],



addSword : function (...) {}

A CLOSER LOOK AT THAT “OR” ASSIGNMENT

The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

```
var armory = { addSword: function (sword) {  
    this.swords = [ ] || this.swords;  
    this.swords.push(sword);  
}  
};
```

This logical assignment will always short-circuit to accept the [], because it is not a “falsy” value.

```
armory.addSword("Broadsword");  
armory.addSword("Katana");
```



addSword : function (...) {}

A CLOSER LOOK AT THAT “OR” ASSIGNMENT

The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

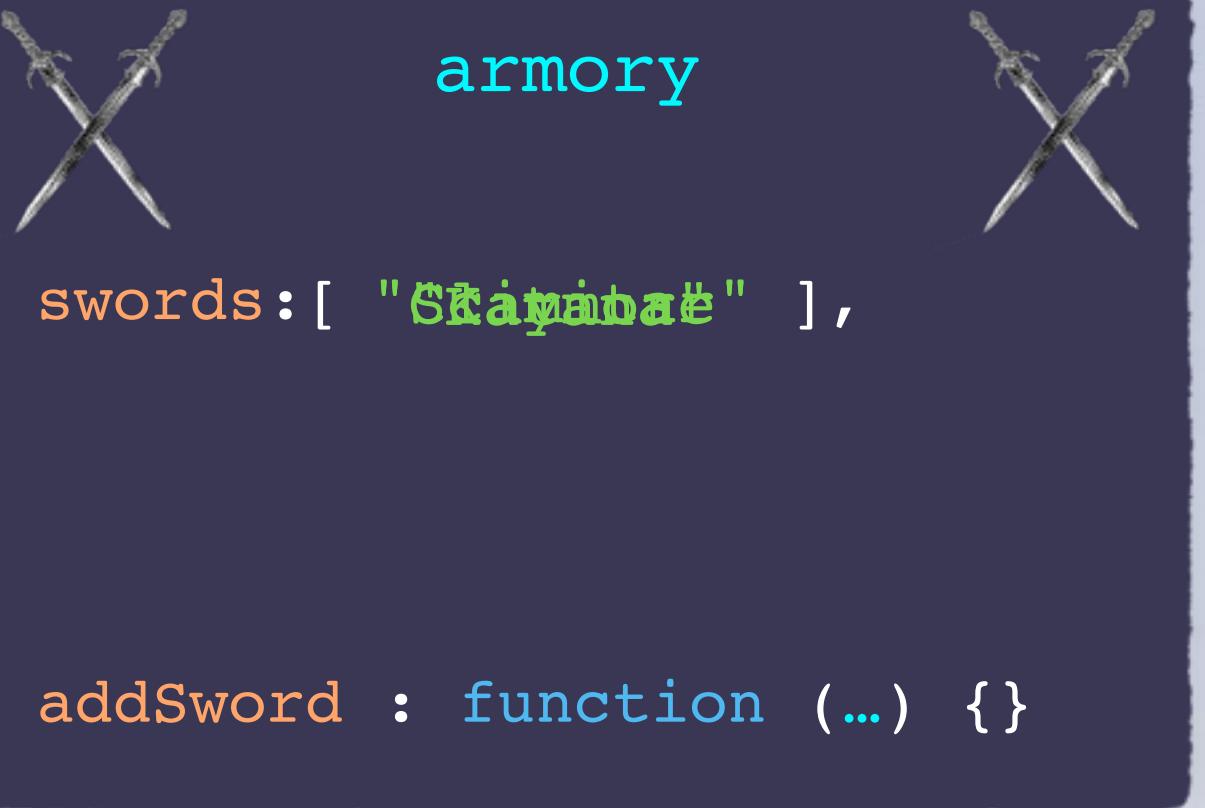
```
var armory = { addSword: function (sword) {  
    this.swords = [ ] || this.swords;  
    this.swords.push(sword);  
}  
};
```

This logical assignment will always short-circuit to accept the [], because it is not a “falsy” value.

```
armory.addSword("Broadsword");  
armory.addSword("Katana");  
armory.addSword("Claymore");  
armory.addSword("Scimitar");
```



```
console.log(armory.swords);  
  
X → [ "Scimitar" ]
```



A CLOSER LOOK AT THAT “OR” ASSIGNMENT

The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

```
var armory = { addSword: function (sword) {  
    this.swords = [ ] || this.swords;  
    this.swords.push(sword);  
}  
};
```

This logical assignment will always short-circuit to accept the [], because it is not a “falsy” value.

```
armory.addSword("Broadsword");  
armory.addSword("Katana");  
armory.addSword("Claymore");  
armory.addSword("Scimitar");
```



```
console.log(armory.swords);
```



→ ["Scimitar"] ←

We should always carefully construct our logical assignments, so that our default cases do not override our desired cases or existing work.

LET'S EXAMINE A FEW OTHER “OR” ASSIGNMENTS

Various arrangements of “truthy” and “falsy” values will yield unique results.

```
var result1 = 42 || undefined;  
console.log(result1);
```

→ 42

Short-circuit, `undefined` is never examined.

```
var result2 = ["Sweet", "array"] || 0;  
console.log(result2);
```

→ ["Sweet", "array"]

Short-circuit, `0` is never examined.

```
var result3 = {type: "ring", stone: "diamond"} || "";  
console.log(result3);
```

→ {type: "ring", stone: "diamond"}

Short-circuit, `""` is never examined.

LET'S EXAMINE A FEW OTHER “OR” ASSIGNMENTS

Various arrangements of “truthy” and “falsy” values will yield unique results.

```
var result1 = 42 || undefined;  
console.log(result1);
```

→ 42

```
var result2 = ["Sweet", "array"] || 0;  
console.log(result2);
```

→ ["Sweet", "array"]

```
var result3 = {type: "ring", stone: "diamond"} || "";  
console.log(result3);
```

→ {type: "ring", stone: "diamond"}

LET'S EXAMINE A FEW OTHER “OR” ASSIGNMENTS

Various arrangements of “truthy” and “falsy” values will yield unique results.

```
var result1 = undefined || 42;  
console.log(result1);
```

→ 42

```
var result2 = 0 || ["Sweet", "array"];  
console.log(result2);
```

→ ["Sweet", "array"]

```
var result3 = "" || {type: "ring", stone: "diamond"};  
console.log(result3);
```

→ {type: "ring", stone: "diamond"}

These logical assignments still accept the first non-falsy value found.

LET'S EXAMINE A FEW OTHER “OR” ASSIGNMENTS

When all elements are “truthy”, we’ll get the FIRST “truthy” value found.

```
var result1 = "King" || "Arthur";  
console.log(result1);
```

→ "King"

```
var result2 = "Arthur" || "King";  
console.log(result2);
```

→ "Arthur"

Since “truthy” values are found first in each, the logical assignments will short-circuit here, too.

LET'S EXAMINE A FEW OTHER “OR” ASSIGNMENTS

When all elements are “falsy”, we’ll get the LAST “falsy” value found.

```
var result1 = undefined || "";  
console.log(result1);
```



```
var result2 = "" || undefined;  
console.log(result2);
```



The compiler continues to search for a “truthy” value, but finding none, assigns the last value encountered.

THE SWORD OF SYNTAX

Section 2
Logical Assignment II:
The “AND” Operator

JAVASCRIPT
BEST PRACTICES

OKAY, SO . . . WHAT ABOUT “AND”?

The `&&` operator takes the rightmost “truthy” value or the first “falsy” value.

```
var result1 = undefined && 42;  
console.log(result1);
```

→ undefined ←

Short-circuit, `42` is never examined.

```
var result2 = 0 && ["Sweet", "array"];  
console.log(result2);
```

→ 0 ←

Short-circuit, the array is never examined.

```
var result3 = "" && {type: "ring", stone: "diamond"};  
console.log(result3);
```

→ "" ←

Short-circuit, the object is never examined.

OKAY, SO . . . WHAT ABOUT “AND”?

The `&&` operator takes the rightmost “truthy” value or the first “falsy” value.

```
var result1 = undefined && 42;  
console.log(result1);
```

→ undefined

```
var result2 = 0 && ["Sweet", "array"];  
console.log(result2);
```

→ 0

```
var result3 = "" && {type: "ring", stone: "diamond"};  
console.log(result3);
```

→ ""

OKAY, SO . . . WHAT ABOUT “AND”?

The `&&` operator takes the rightmost “truthy” value or the first “falsy” value.

```
var result1 = undefined && 42;  
console.log(result1);
```

```
var result2 = 0 && ;  
console.log(result2);
```

```
var result3 = "" && {type: "ring", stone: "diamond"};  
console.log(result3);
```

OKAY, SO . . . WHAT ABOUT “AND”?

The `&&` operator takes the rightmost “truthy” value or the first “falsy” value.

```
var result1 = 42 && undefined;  
console.log(result1);
```

→ undefined

```
var result2 = ["Sweet"] && ;  
console.log(result2);
```

→ 0

```
var result3 = {type: "ring", stone: "diamond"} && ""};  
console.log(result3);
```

→ ""

The compiler seeks to verify that only “truthy” values exist, but if it ever arrives at a “falsy” value, that value gets the assignment.

OKAY, SO . . . WHAT ABOUT “AND”?

The `&&` operator takes the rightmost “truthy” value or the first “falsy” value.

```
var result1 =      &&           ;  
console.log(result1);
```

```
var result2 =           &&     ;  
console.log(result2);
```

OKAY, SO . . . WHAT ABOUT “AND”?

When all elements are “truthy”, `&&` will return the LAST “truthy” value found.

```
var result1 = "King" && "Arthur";  
console.log(result1);
```

→ Arthur

```
var result2 = "Arthur" && "King";  
console.log(result2);
```

→ King

```
var result1 = "King" || "Arthur";  
console.log(result1);
```

→ King

```
var result2 = "Arthur" || "King";  
console.log(result2);
```

→ Arthur

Once all values are verified to be “truthy”, the logical assignment returns the last value encountered.

For comparison, check out the OR results, which we’ve already seen. The first “truthy” value encountered gets returned.

OKAY, SO . . . WHAT ABOUT “AND”?

When all elements are “falsy”, AND will return the FIRST “falsy” value found.

```
var result1 = undefined && "";
console.log(result1);
```

→ undefined

```
var result2 = "" && undefined;
console.log(result2);
```

→ ""

```
var result1 = undefined || "";
console.log(result1);
```

→ ""

```
var result2 = "" || undefined;
console.log(result2);
```

→ undefined

As you might expect, the first “falsy” is returned in short-circuit style.

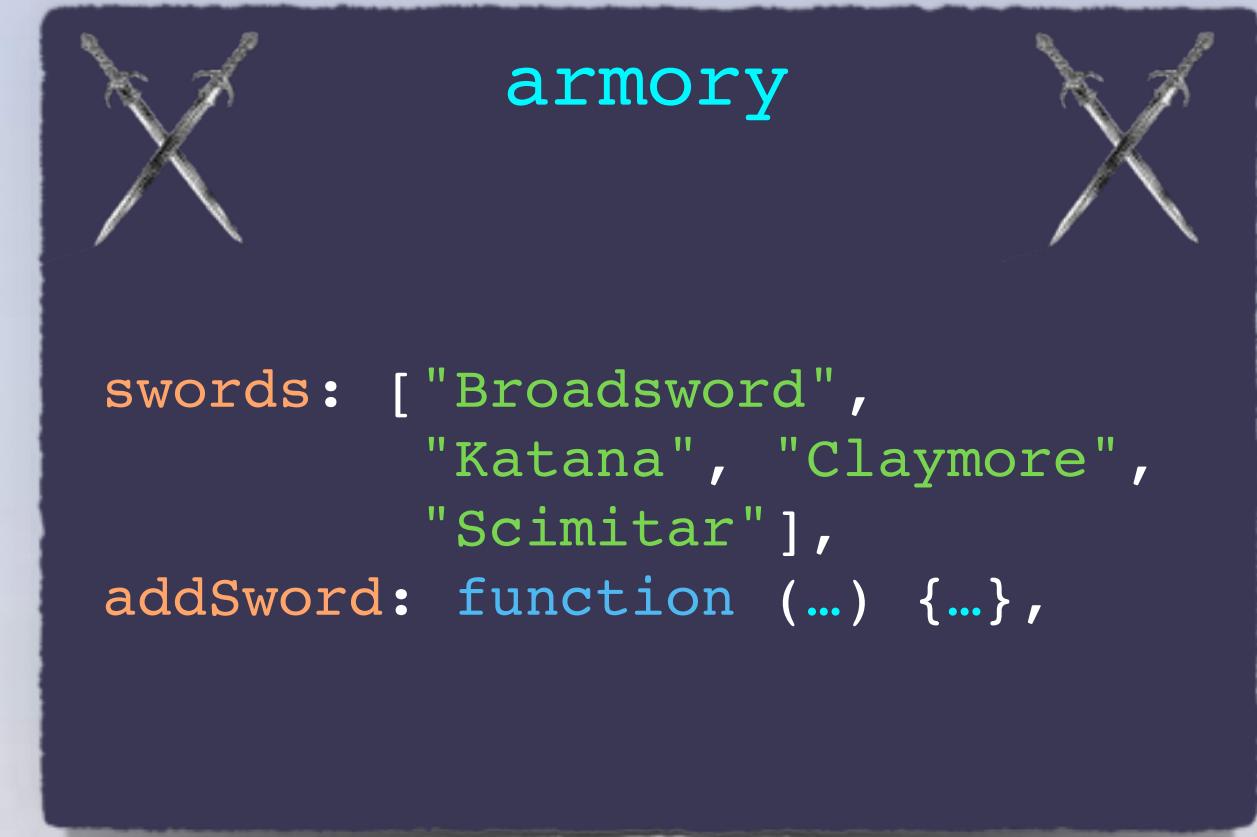
Again, compare that with OR, which still tries to find a “truthy” value somewhere before finally returning a “falsy” value as a last resort...

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
};
```

In our new function, we'll return the results of a ternary conditional.



“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
};
```

We first check to see if the `swords` array actually holds the sword being asked for.



```
armory.swords.indexOf("Claymore");
```

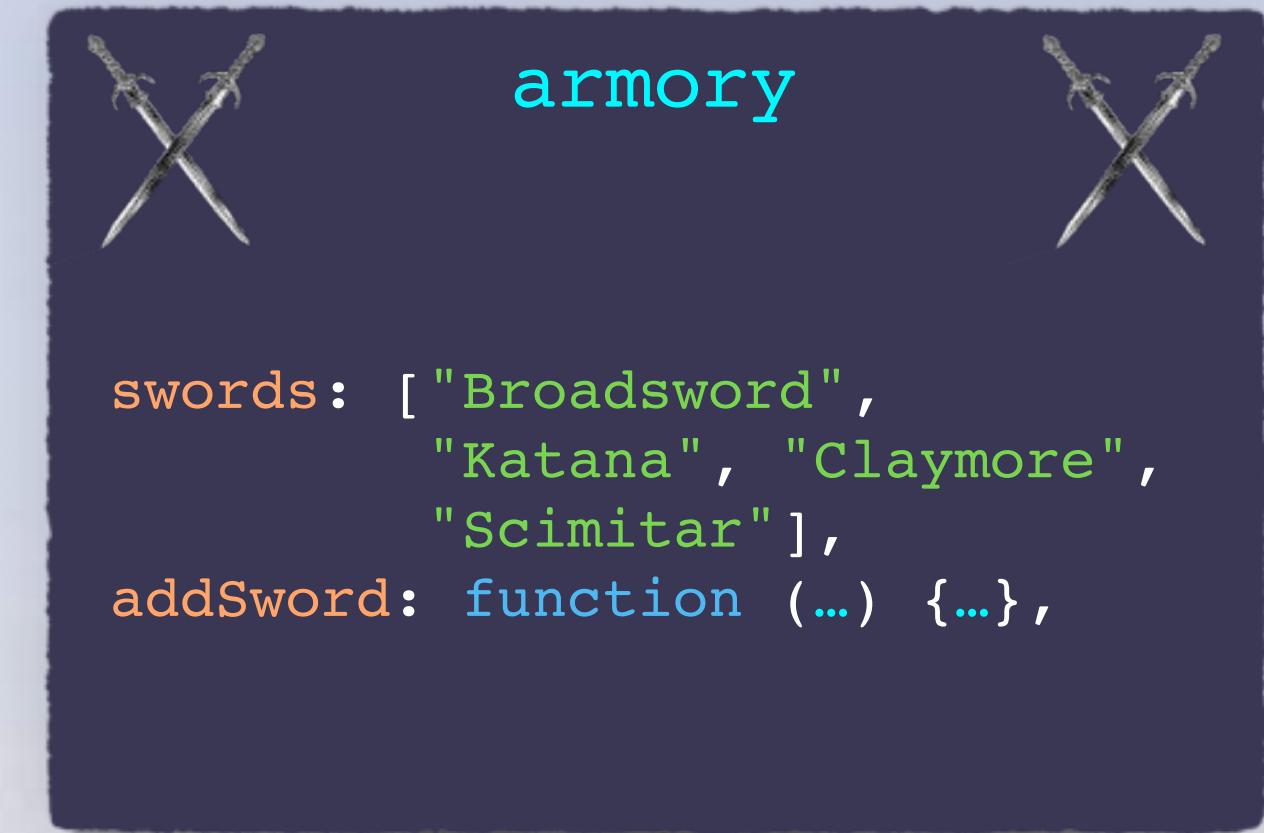
→ 2



```
armory.swords.indexOf("Donut");
```

→ -1

The array prototype's `indexOf` method will return the first found index of a value, or `-1` if it's not found. Thus, with a check for a number zero or greater, our ternary can verify the sword is present.



“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

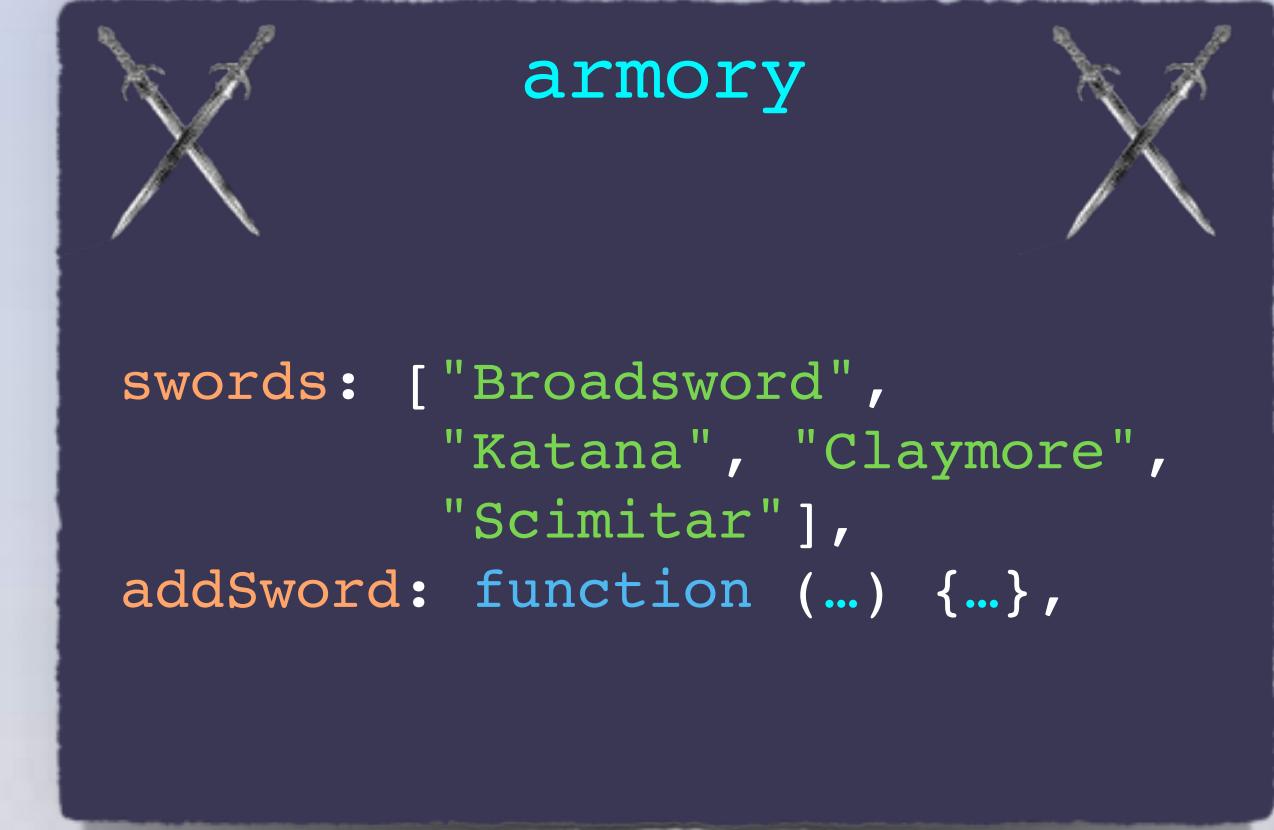
```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    };
```

If the right sword is available, we will remove it from the swords array with some index cleverness.

```
var soldiers = ["Knights", "Pikemen", "Archers"];  
soldiers.splice( 1, 2 );
```



The `splice` method can remove elements anywhere in an array, by first passing in the index you'd like to start at, followed by how many items you'd like to remove.



“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    };
```

If the right sword is available, we will remove it from the swords array with some index cleverness.

```
var soldiers = ["Knights", "Pikemen", "Archers"];  
soldiers.splice( 1, 2 );
```

Goes to index 1,
and cuts out
two entries.

```
armory  
swords: [ "Broadsword",  
          "Katana", "Claymore",  
          "Scimitar" ],  
addSword: function ( ... ) { ... },
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    };
```

If the right sword is available, we will remove it from the swords array with some index cleverness.

```
var soldiers = ["Knights", "Pikemen", "Archers"];  
soldiers.splice( 1, 2 );
```

→ ["Pikemen", "Archers"]



Returns the values in their own array.

 **armory** 

```
swords: [ "Broadsword",  
          "Katana", "Claymore",  
          "Scimitar" ],  
addSword: function ( ... ) { ... },
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    };
```

So we'll find the location of the first matching sword available using `indexOf`, and then take just one entry out.

```
var soldiers = ["Knights", "Pikemen", "Archers"];  
soldiers.splice( 1, 2 );
```

→ ["Pikemen", "Archers"]



Returns the values in their own array.

 **armory** 

```
swords: [ "Broadsword",  
          "Katana", "Claymore",  
          "Scimitar" ],  
addSword: function ( ... ) { ... },
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    };
```

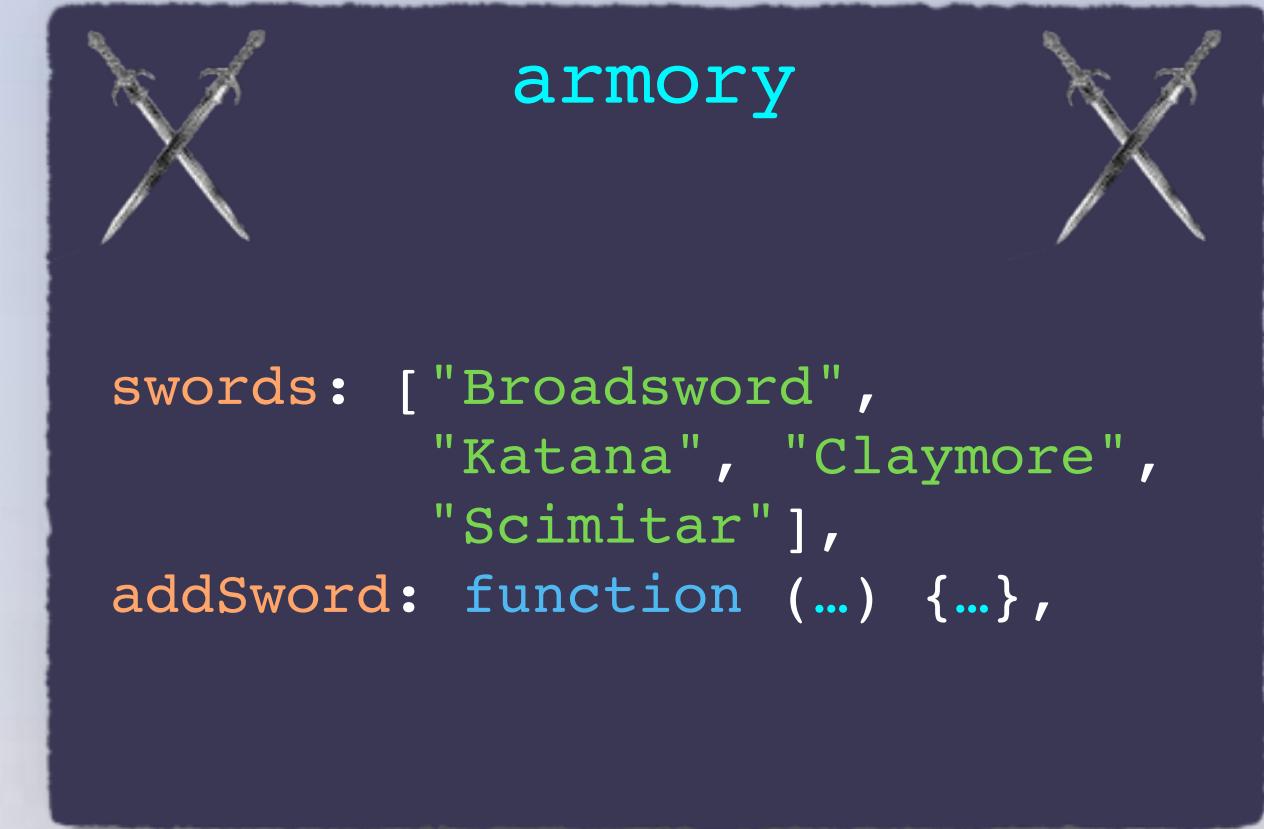
Since we get an array back, we'll access the zeroth index of that array to get just the sword's string.

```
var soldiers = ["Knights", "Pikemen", "Archers"];  
soldiers.splice( 1, 2 );
```

→ ["Pikemen", "Archers"]



Returns the values in their own array.



```
swords: ["Broadsword",  
"Katana", "Claymore",  
"Scimitar"],  
addSword: function (...) {...},  
armory
```

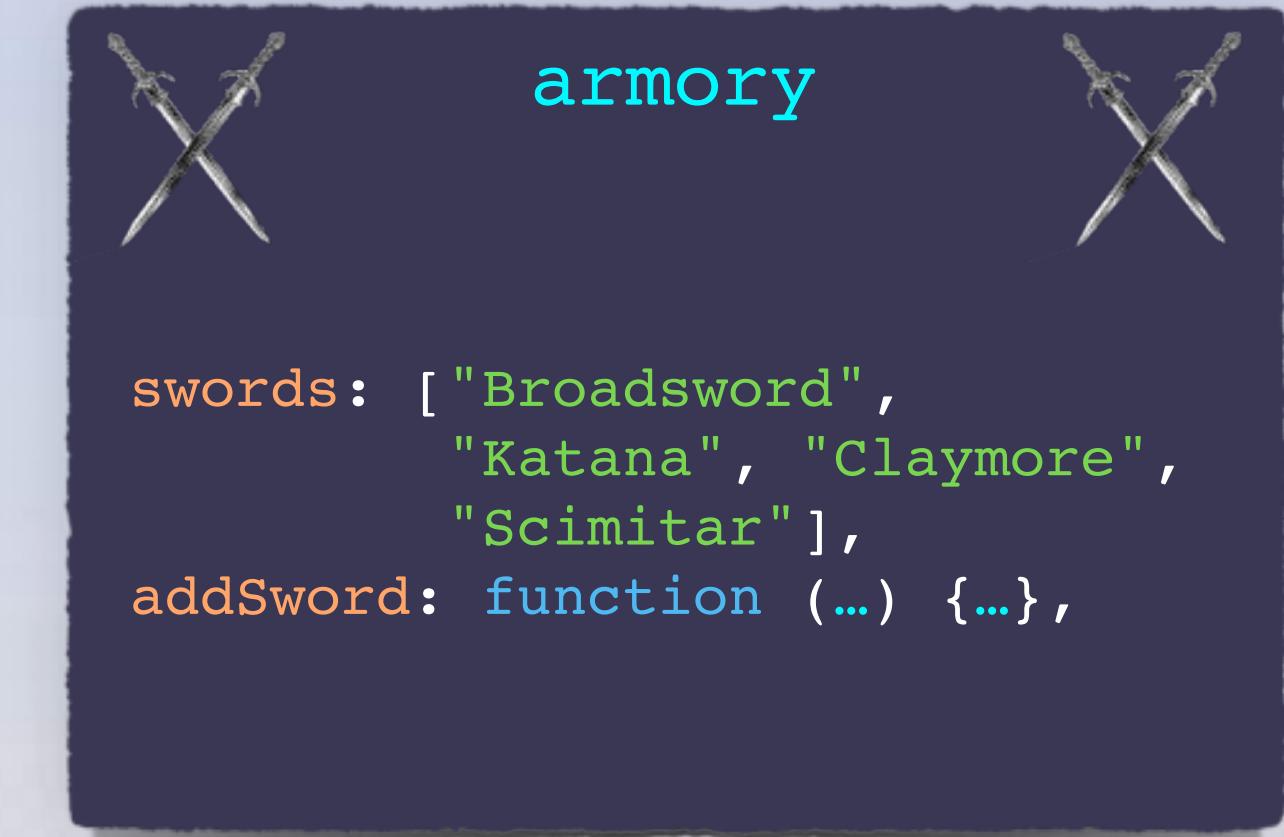
“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```



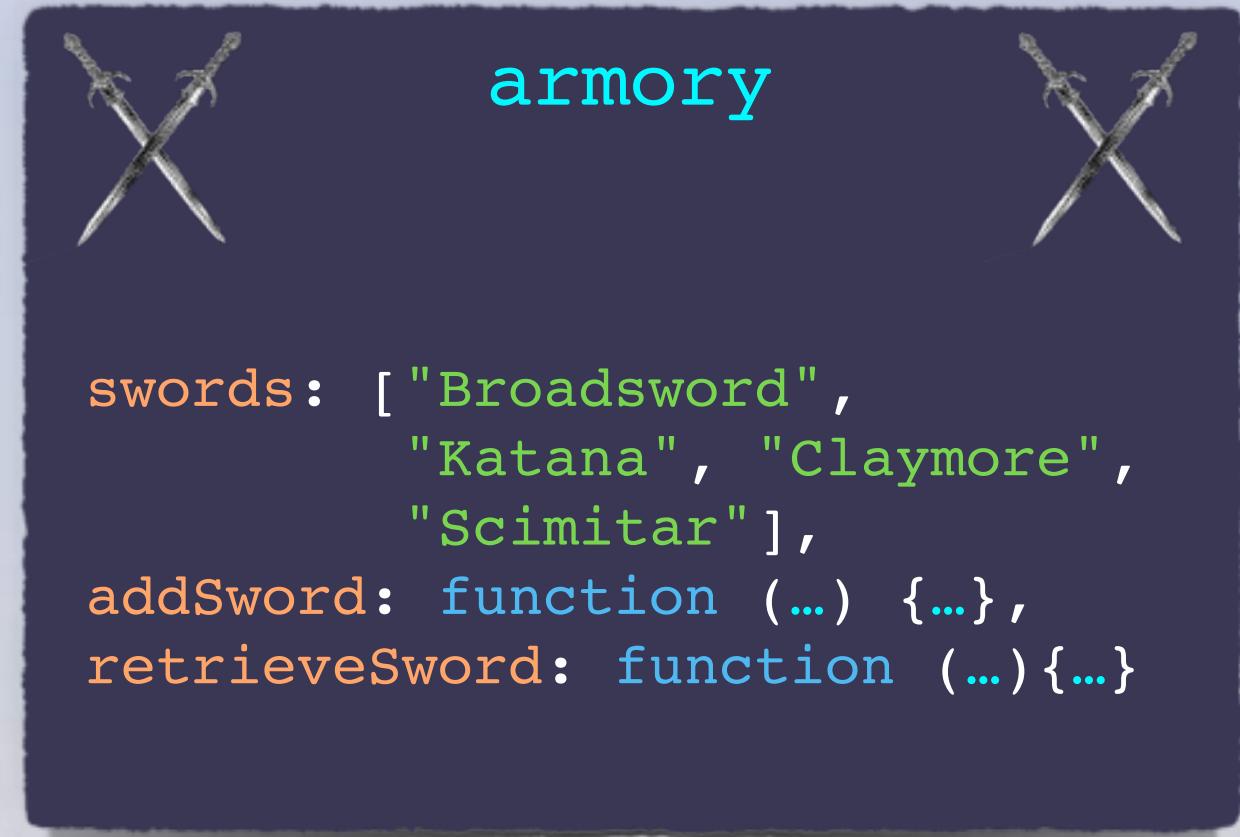
If we never initially found the sword, however, we'll alert that this particular sharp thing ain't here. This will be useful when we employ `&&`.



“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```



“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

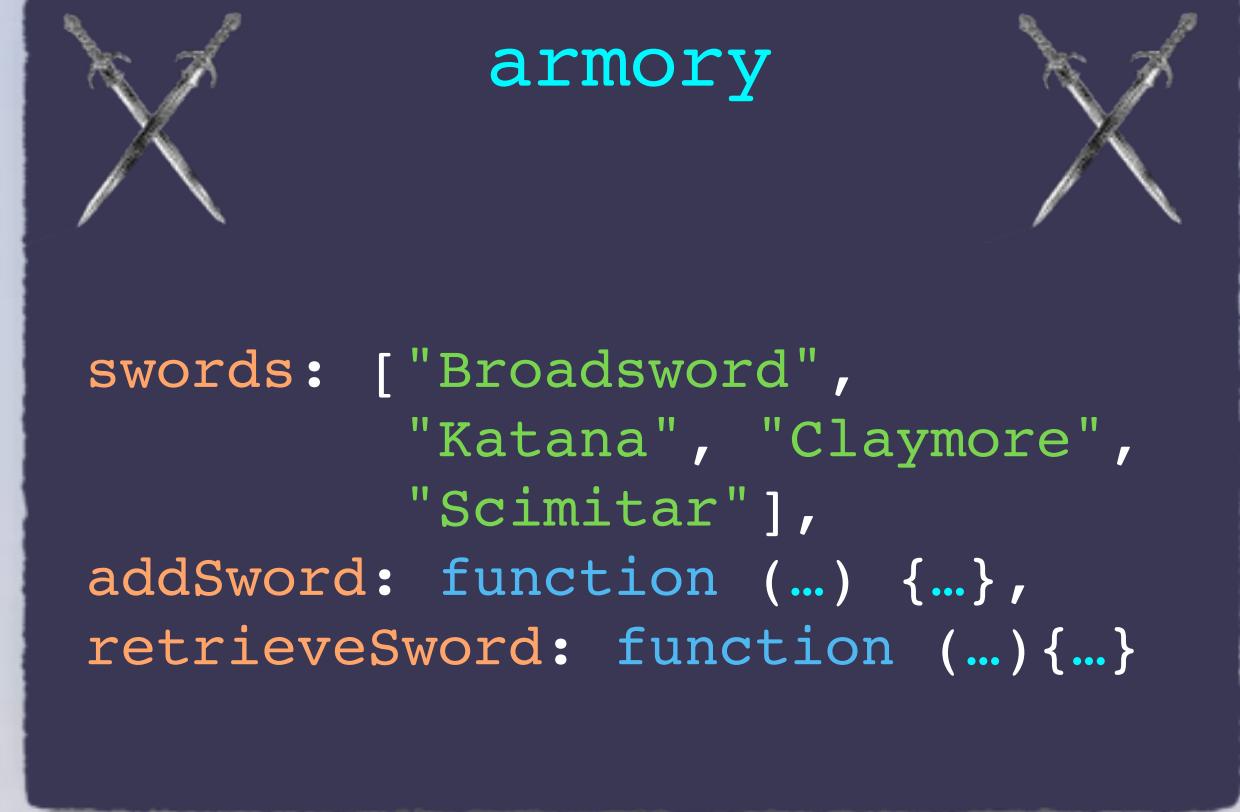
The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
  return (this.swords.indexOf(request) >= 0) ?  
    this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    alert("No " + request + ", baby!");  
};
```

```
var isKnight = true;
```



We only want to let Knights retrieve weapons from the `armory` ... no serial killers, please.



```
armory  
swords: ["Broadsword",  
         "Katana", "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```



Welp, `isKnight` is currently true, so `&&` will keep verifying ...



armory



```
swords: ["Broadsword",  
         "Katana", "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
  return (this.swords.indexOf(request) >= 0) ?  
    this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    alert("No " + request + ", baby!");  
};
```

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```



If the result of this function call is a string (and thus “truthy”), we’ll get precisely the sword we’re looking for in our assignment.

armory

```
swords: ["Broadsword",  
         "Katana", "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

First, is there even
a Katana in the
swords array?



```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```

 **armory** 

```
swords: ["Broadsword",  
         "Katana", "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```

So now we splice the **swords** array, which gives us back another array with the Katana in it. This also removes the Katana from **swords**.

```
armory  
swords: ["Broadsword",  
         "Katana", "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

["Katana"]

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```

So now we splice the **swords** array, which gives us back another array with the Katana in it. This also removes the Katana from **swords**.



```
swords: [ "Broadsword",  
          "Claymore",  
          "Scimitar" ],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

["Katana"]

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```

So now we splice the **swords** array, which gives us back another array with the Katana in it. This also removes the Katana from **swords**.



armory

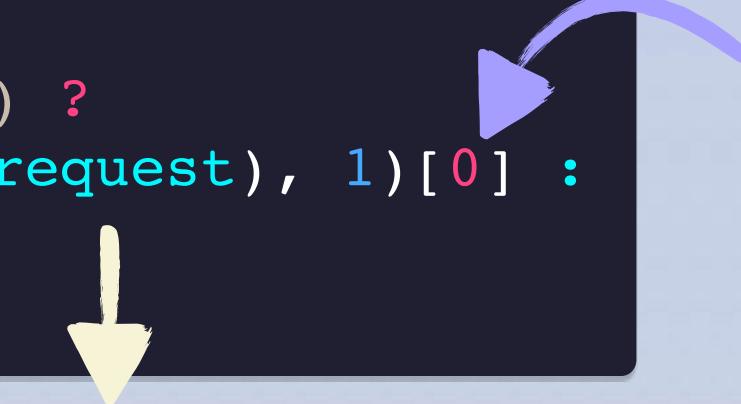


```
swords: [ "Broadsword",  
          "Claymore",  
          "Scimitar" ],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```



["Katana"]

Lastly, we access the first index of the returned array, so that we can have the Katana as a returned string.

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```



```
swords: [ "Broadsword",  
          "Claymore",  
          "Scimitar" ],  
addSword: function (...){...},  
retrieveSword: function (...){...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```



Lastly, we access the first index of the returned array, so that we can have the Katana as a returned string.

"Katana"

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```



armory

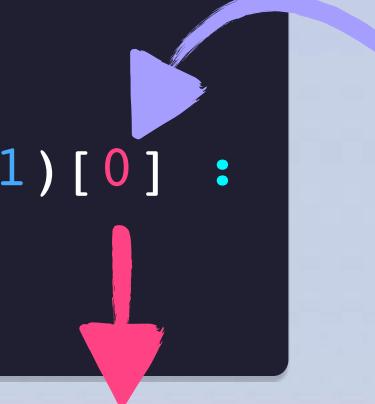


```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

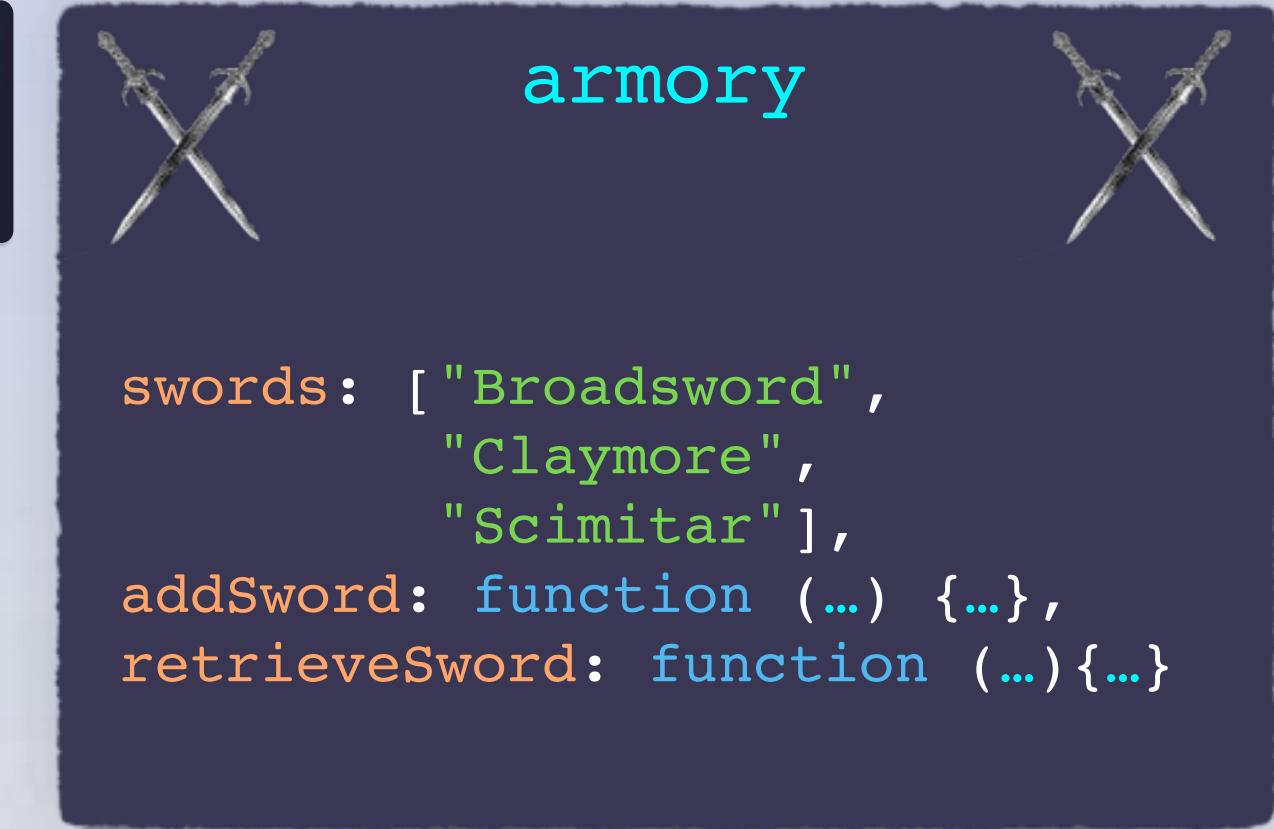
```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```



"Katana"

Lastly, we access the first index of the returned array, so that we can have the Katana as a returned string.

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```

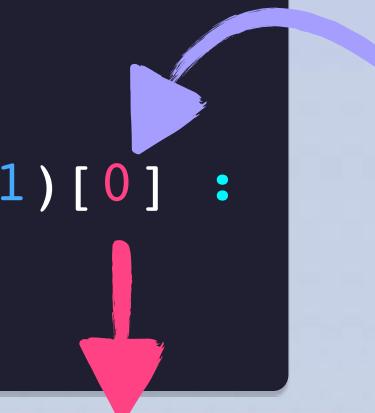


```
armory  
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```



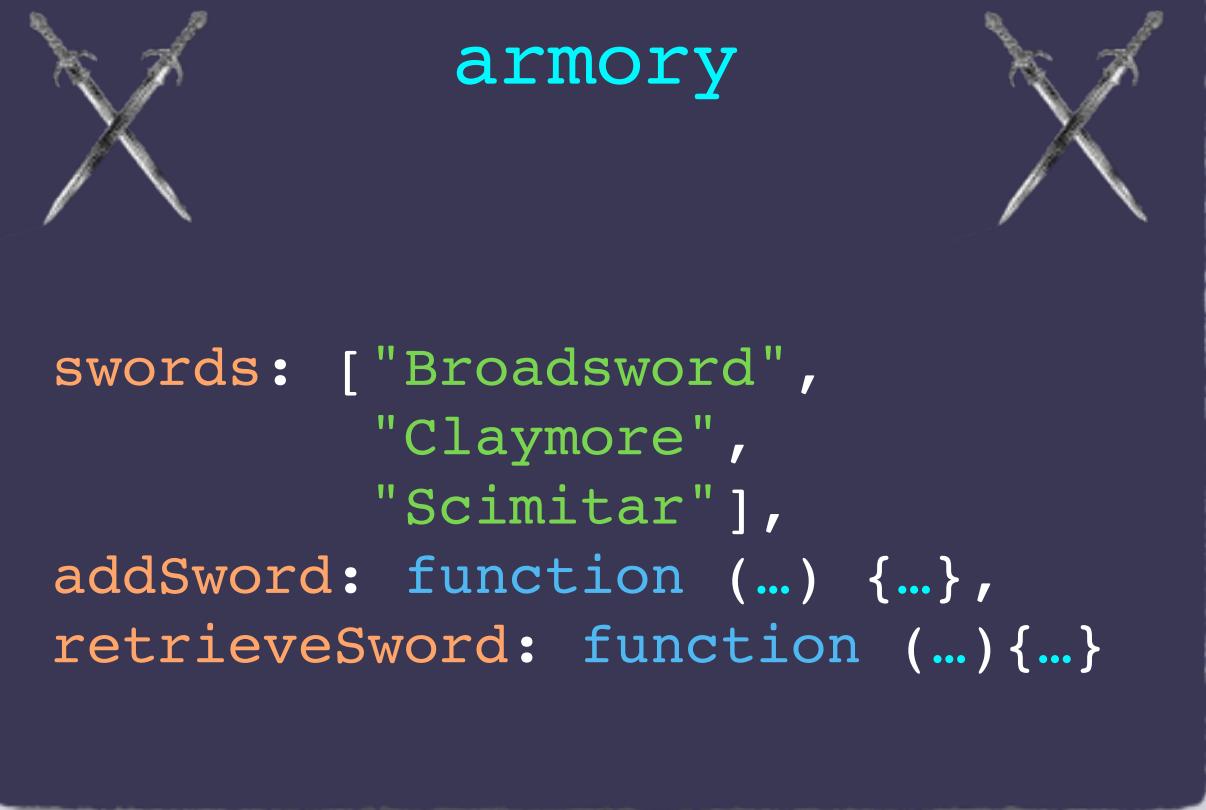
"Katana"

Lastly, we access the first index of the returned array, so that we can have the Katana as a returned string.

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```



`&&` returns the last present value when all values are “truthy.”



“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");
```



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Katana");  
console.log(weapon);
```

→ Katana



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

First, is there even a Rapier in the `swords` array? Nope.



```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Rapier");  
console.log(weapon);
```



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

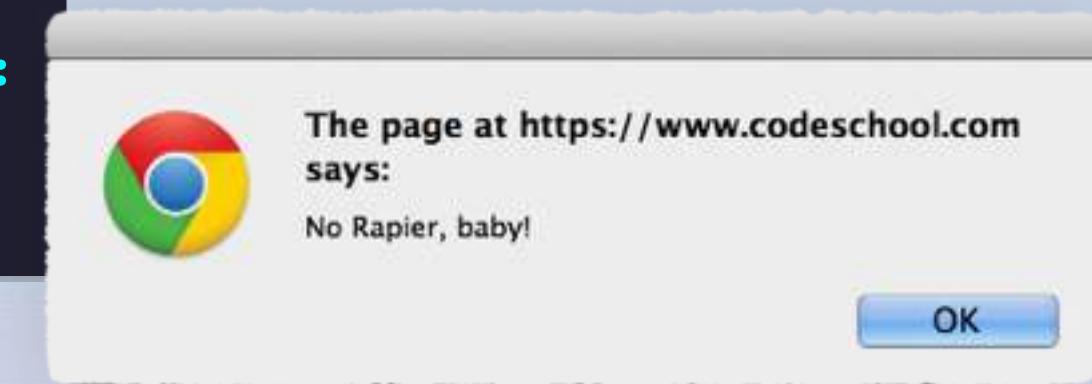


So, we'll alert the message
that no rapier exists.

```
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword("Rapier");  
console.log(weapon);
```

→ undefined

The `alert` does not return a value after
its execution, so `weapon` will receive
`undefined` as an assignment.



```
armory  
  swords: ["Broadsword",  
           "Claymore",  
           "Scimitar"],  
  addSword: function (...){...},  
  retrieveSword: function (...){...}
```

“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

The AND logical operator lets us to check multiple conditions before allowing assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var isKnight = false;  
var weapon = isKnight && armory.retrieveSword("Rapier");  
console.log(weapon);
```



→ false

As soon as `&&` spots that `isKnight` is now `false`, it stops trying to verify everything is `true`, and makes the `false` assignment. This value can now be used in a conditional check elsewhere.

armory

```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...){...},  
retrieveSword: function (...){...}
```

LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var isKnight = ;  
var weapon = isKnight && armory.retrieveSword();  
console.log(weapon);
```



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var armoryIsOpen = true;  
var isKnight = true;  
var weapon = isKnight && armory.retrieveSword();  
  
console.log(weapon);
```



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var armoryIsOpen = true;  
var isKnight = true;  
var weapon = isKnight &&  
            armory.retrieveSword();  
console.log(weapon);
```



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var armoryIsOpen = true;  
var isKnight = true;  
var weapon = armoryIsOpen && isKnight &&  
    armory.retrieveSword("Claymore");  
console.log(weapon);
```



armory



```
swords: ["Broadsword",  
         "Claymore",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var armoryIsOpen = true;  
var isKnight = true;  
var weapon = armoryIsOpen && isKnight &&  
    armory.retrieveSword("Claymore");  
console.log(weapon);
```



armory



```
swords: ["Broadsword",  
         "Scimitar"],  
addSword: function (...) {...},  
retrieveSword: function (...) {...}
```

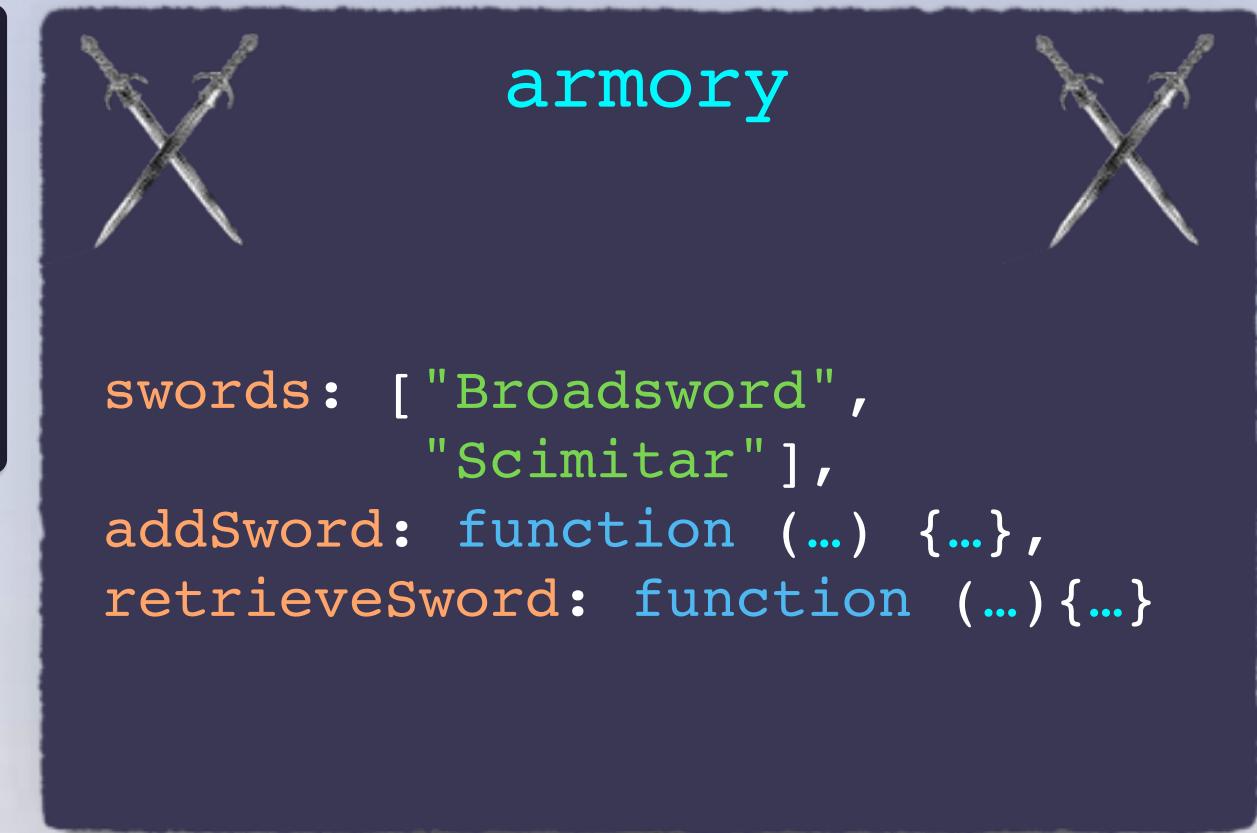
LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
    return (this.swords.indexOf(request) >= 0) ?  
        this.swords.splice(this.swords.indexOf(request), 1)[0] :  
        alert("No " + request + ", baby!");  
};
```

```
var armoryIsOpen = true;  
var isKnight = true;  
var weapon = armoryIsOpen && isKnight &&  
    armory.retrieveSword("Claymore");  
console.log(weapon);
```

→ Claymore



LASTLY, LOGICAL ASSIGNMENTS ARE EXTENDABLE

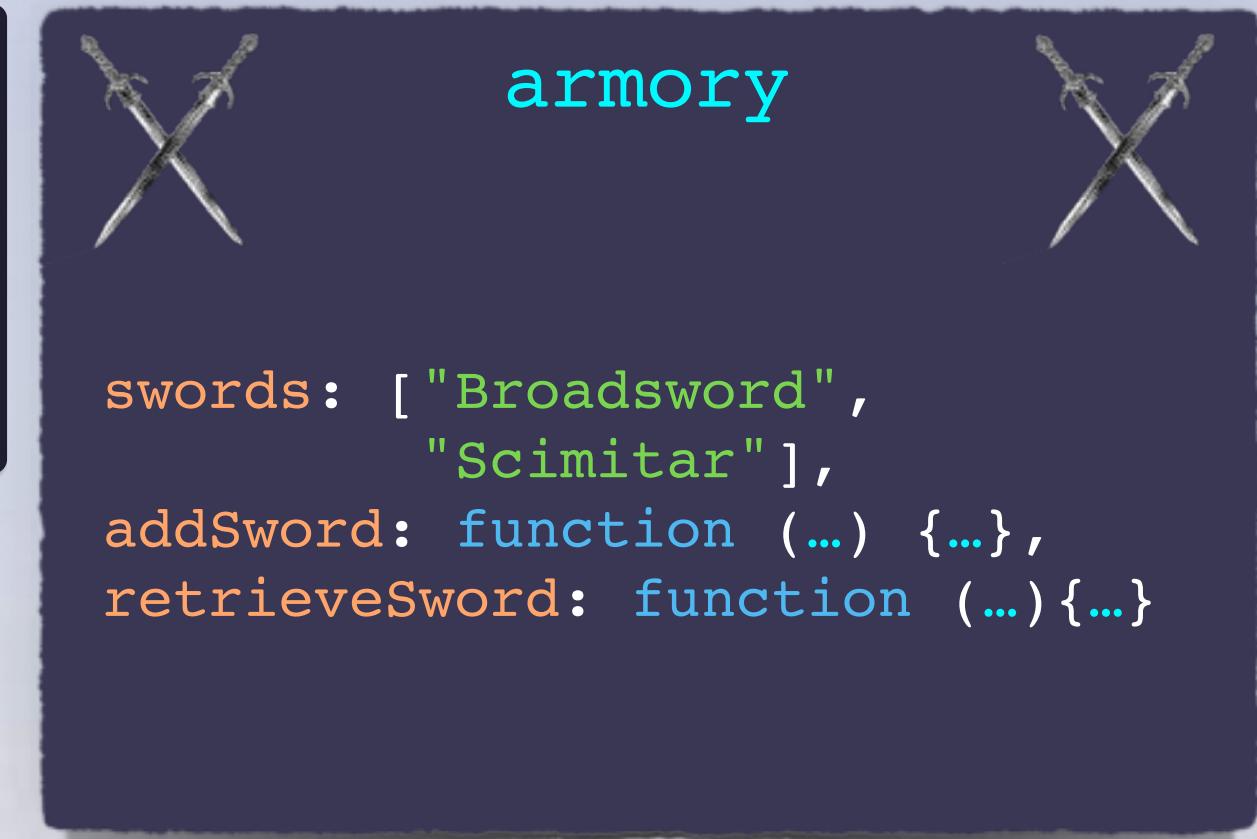
We can string together as many operators as is necessary to choose the right assignment.

```
armory.retrieveSword = function (request){  
  return (this.swords.indexOf(request) >= 0) ?  
    this.swords.splice(this.swords.indexOf(request), 1)[0] :  
    alert("No " + request + ", baby!");  
};
```

```
var armoryIsOpen = false;  
var isKnight = true;  
var weapon = armoryIsOpen && isKnight &&  
  armory.retrieveSword("Claymore");  
console.log(weapon);
```

As before, when `&&` spots that `armoryIsOpen` is `false`, there's no need to go further, and the assignment is made.

→ false



THE SWORD OF SYNTAX

Section 3
The Switch Block

JAVASCRIPT
BEST PRACTICES

A CONDITIONAL FOR MULTIPLE POSSIBILITIES

JavaScript has an alternate way of taking action based on actual values, not just Booleans.



1

Broadsword



2

Claymore



3

Longsword



4

Mace



5

War Hammer



6

Battle Axe



7

Halberd

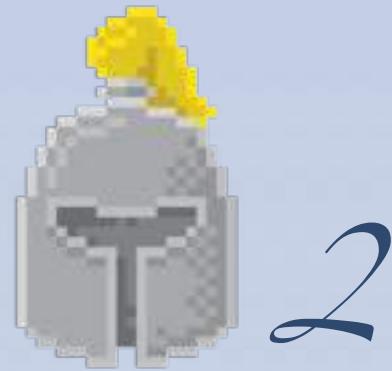


8

Morning Star

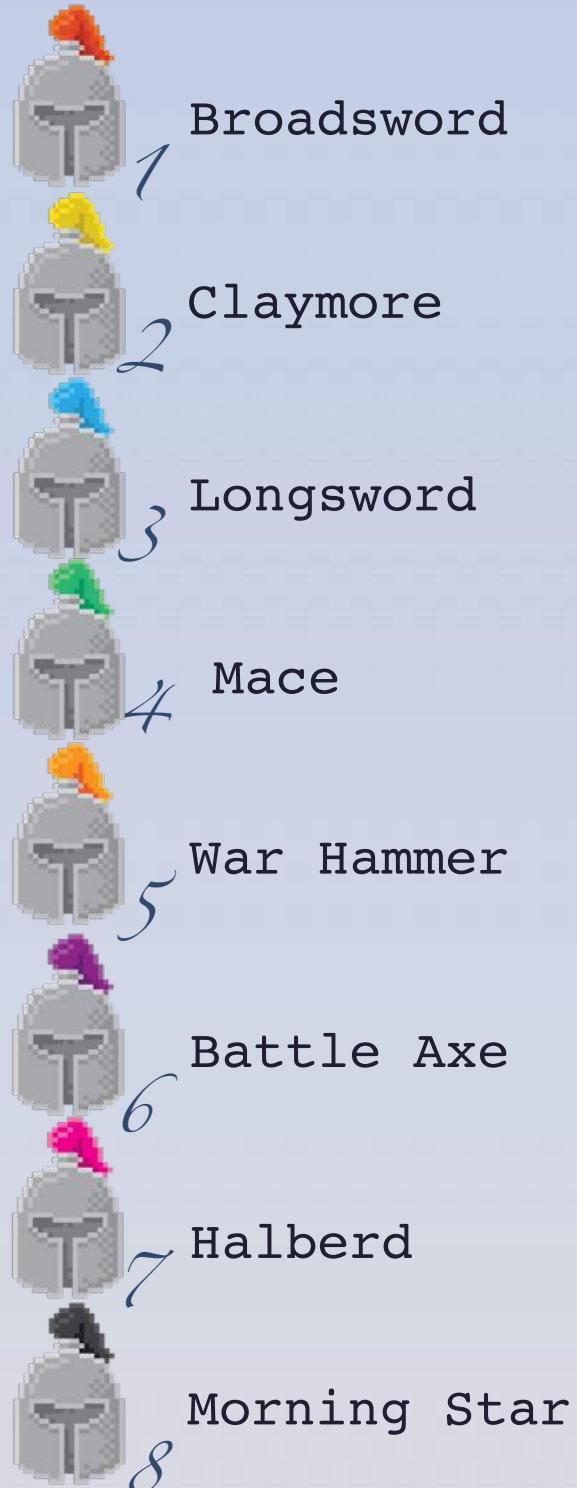
A CONDITIONAL FOR MULTIPLE POSSIBILITIES

JavaScript has an alternate way of taking action based on actual values, not just Booleans.



A CONDITIONAL FOR MULTIPLE POSSIBILITIES

JavaScript has an alternate way of taking action based on actual values, not just Booleans.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    if(regiment == 1) {  
        this.weapon = "Broadsword";  
    } else if(regiment == 2) {  
        this.weapon == "Claymore";  
    } else if(regiment == 3) {  
        this.weapon = "Longsword";  
    } else if(regiment == 4) {  
        this.weapon = "Mace";  
    } else if(regiment == 5) {  
        this.weapon = "War Hammer";  
    } else if(regiment == 6) {  
        this.weapon = "Battle Axe";  
    } else if(regiment == 7) {  
        this.weapon = "Halberd";  
    } else if(regiment == 8) {  
        this.weapon = "Morning Star";  
    }  
}
```

```
var soldier = new Knight("Timothy", 2);  
console.log(soldier.weapon);
```

→ Claymore

Whereas we currently check repeatedly for a match using **if** and **else**, accessing **regiment** over and over again, a cool keyword can instead jump quickly to a match and take action.

ENTER THE SWITCH KEYWORD

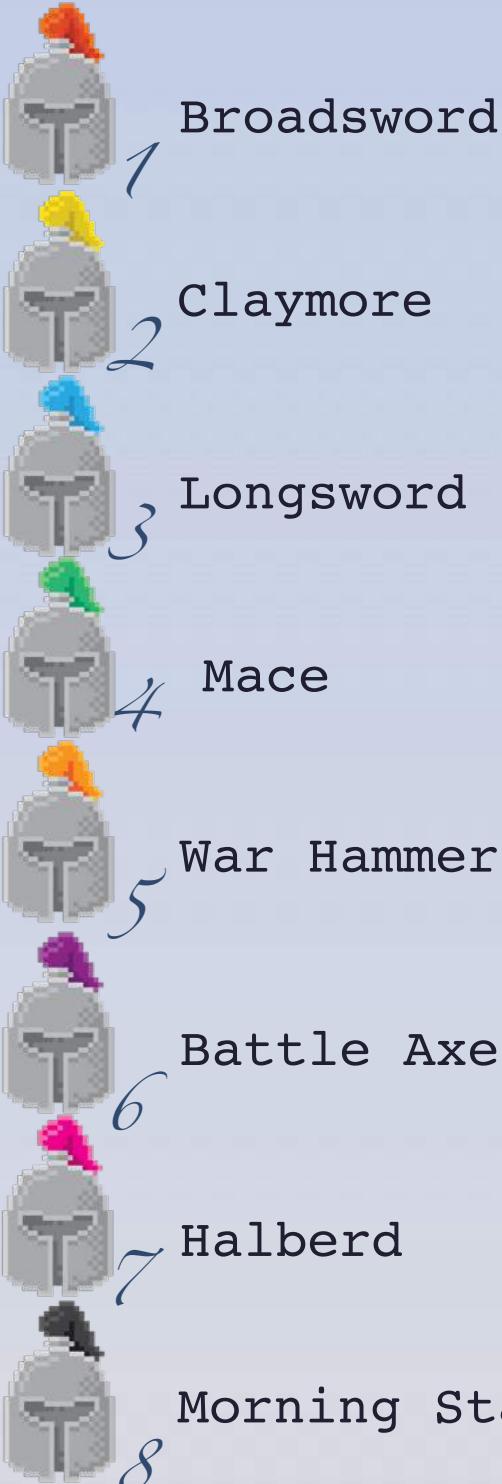
Switch will allow us to assign a weapon based on a value itself, without a Boolean check.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;
```

ENTER THE SWITCH KEYWORD

Switch will allow us to assign a weapon based on a value itself, without a Boolean check.

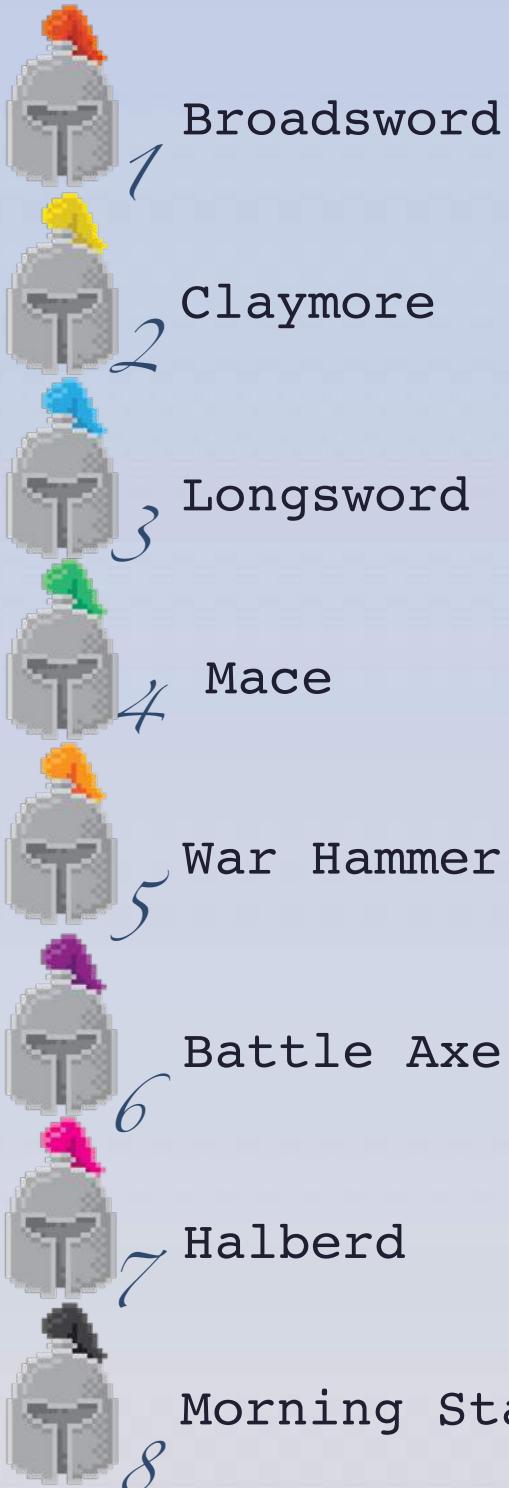


```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
            this.weapon = "Halberd";  
            break;  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```

switch signals to take some unique action based on the value of regiment.

ENTER THE SWITCH KEYWORD

Switch will allow us to assign a weapon based on a value itself, without a Boolean check.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
    }  
}
```

The `switch` block contains multiple `case`'s, where the `case` keyword is followed by one of the possible or desired values of `regiment`.

ENTER THE SWITCH KEYWORD

Switch will allow us to assign a weapon based on a value itself, without a Boolean check.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1: this.weapon = "Broadsword";  
        case 2:  
        case 3:  
        case 4:  
        case 5:  
        case 6:  
        case 7:  
        case 8:  
    }  
}
```

Next, a colon separates the `case` keyword and its value from the appropriate action to take for that value.

ENTER THE SWITCH KEYWORD

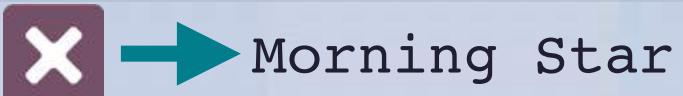
Switch will allow us to assign a weapon based on a value itself, without a Boolean check.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

Um. Uh oh.
So, what happened?



JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

Think of the case keyword and its value as just a label for an executable “starting point”.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

Once a `case` label has been selected by the `switch` block, all other labels are thereafter ignored, because the `switch` jump has already occurred.

```
soldier2  
name: "Richard",  
regiment: 4,  
weapon: "Mace"
```

JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

Think of the case keyword and its value as just a label for an executable “starting point”.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

```
soldier2  
name: "Richard",  
regiment: 4,  
weapon: "War Hammer"
```

JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

Think of the case keyword and its value as just a label for an executable “starting point”.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

```
soldier2  
name: "Richard",  
regiment: 4,  
weapon: "Battle Axe"
```

JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

Think of the case keyword and its value as just a label for an executable “starting point”.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

```
soldier2  
name: "Richard",  
regiment: 4,  
weapon: "Halberd"
```

JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

Think of the case keyword and its value as just a label for an executable “starting point”.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

soldier2
name: "Richard",
regiment: 4,
weapon: "Morning Star" X

THE BREAK KEYWORD PROVIDES A QUICK EXIT

'Break' will allow us to leave the entire block of code that contains it, in this case, the switch.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

```
var soldier2 = new Knight("Richard", 4);  
console.log(soldier2.weapon);
```

```
soldier2  
name: "Richard",  
regiment: 4,  
weapon: "Morning Star" X
```

THE BREAK KEYWORD PROVIDES A QUICK EXIT

'Break' will allow us to leave the entire block of code that contains it, in this case, the switch.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

THE BREAK KEYWORD PROVIDES A QUICK EXIT

‘Break’ will allow us to leave the entire block of code that contains it, in this case, the switch.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
        case 2:  
            this.weapon = "Claymore";  
        case 3:  
            this.weapon = "Longsword";  
        case 4:  
            this.weapon = "Mace";  
        case 5:  
            this.weapon = "War Hammer";  
  
        case 6:  
            this.weapon = "Battle Axe";  
        case 7:  
            this.weapon = "Halberd";  
        case 8:  
            this.weapon = "Morning Star";  
    }  
}
```

THE BREAK KEYWORD PROVIDES A QUICK EXIT

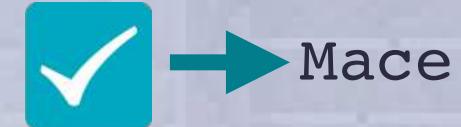
'Break' will allow us to leave the entire block of code that contains it, in this case, the switch.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
    }  
}
```

```
case 6:  
    this.weapon = "Battle Axe";  
    break;  
case 7:  
    this.weapon = "Halberd";  
    break;  
case 8:  
    this.weapon = "Morning Star";  
    break;  
}  
}
```

```
var soldier = new Knight("Richard", 4);  
console.log(soldier.weapon);
```

By exiting the `switch` block immediately, each `break` statement has the effect of ensuring that one and only one `case` action is taken.



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

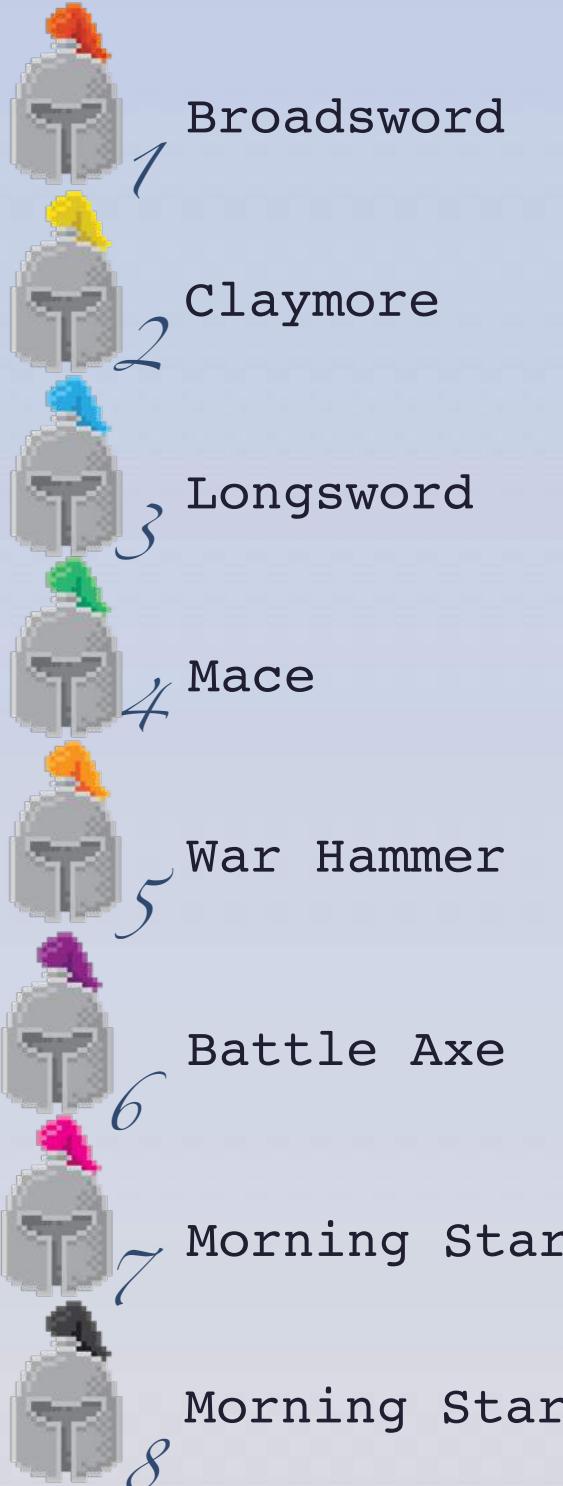
Fall-through will tighten up our code if we stack similar cases before their appropriate action.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
            this.weapon = "Halberd";  
            break;  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```

WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

Fall-through will tighten up our code if we stack similar cases before their appropriate action.

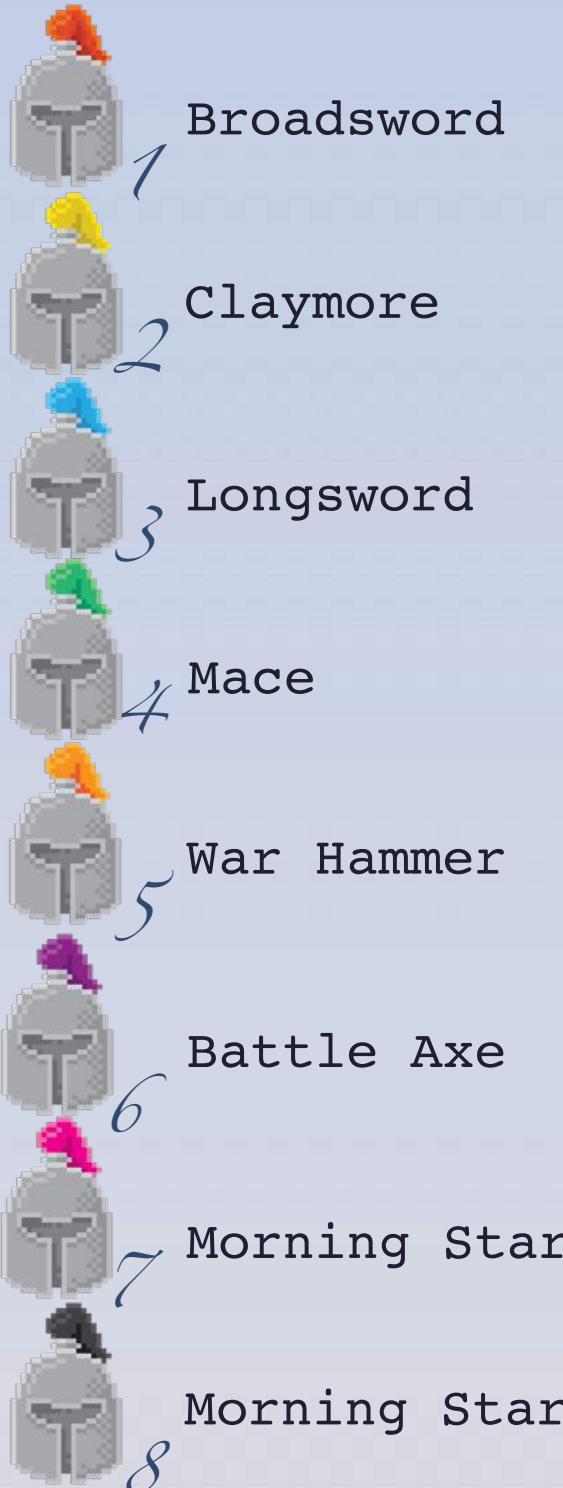


```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
            this.weapon = "Halberd";  
            break;  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

Fall-through will tighten up our code if we stack similar cases before their appropriate action.

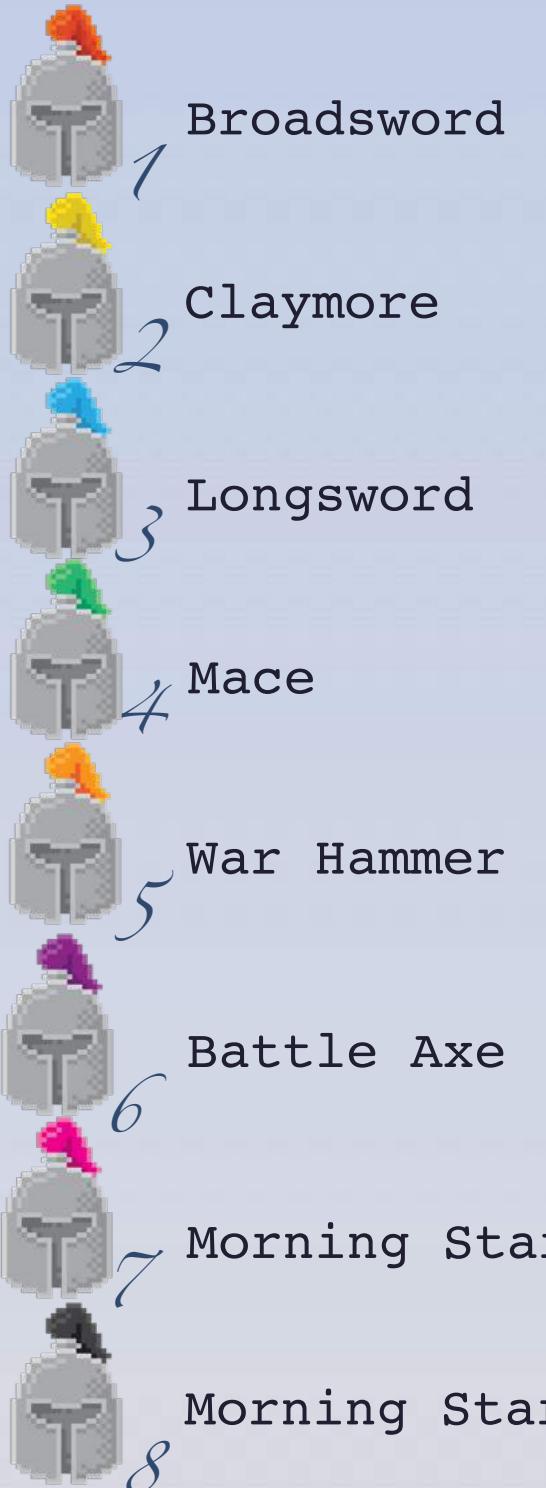


```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

Fall-through will tighten up our code if we stack similar cases before their appropriate action.



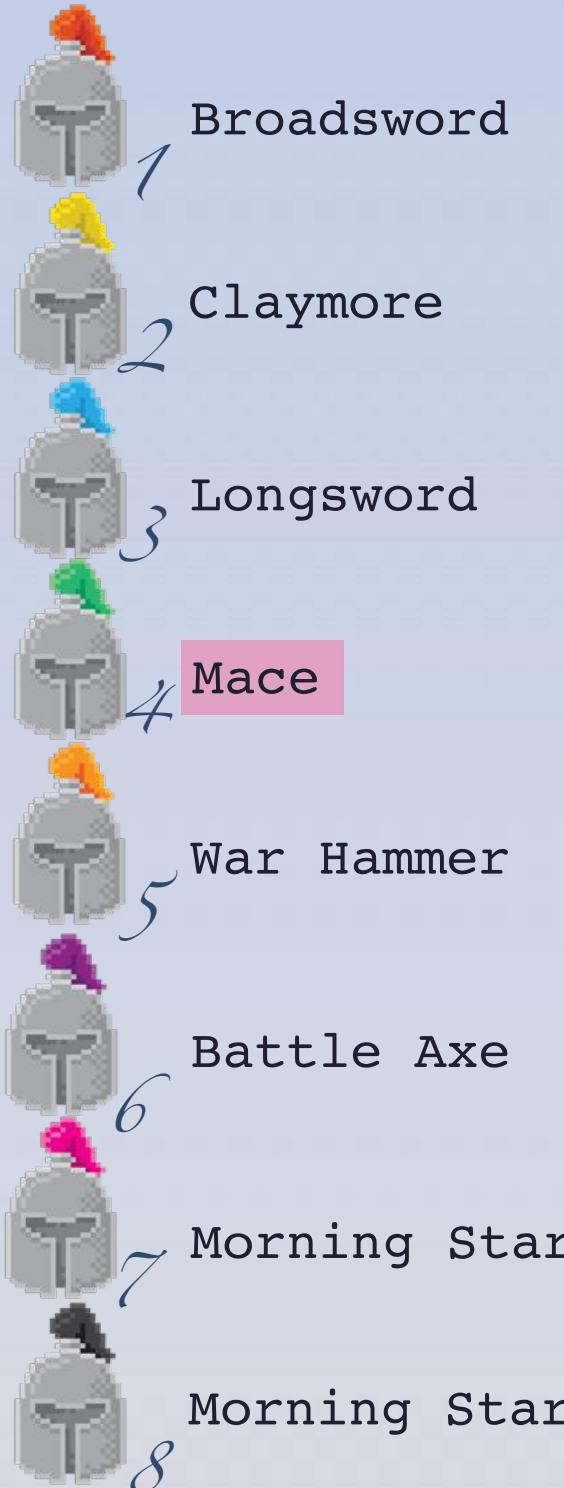
```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```

Stacking cases associates the subsequent actions with all of those cases.



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

JavaScript doesn't care about case order, so we can also move around our numerical cases.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

JavaScript doesn't care about case order, so we can also move around our numerical cases.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
            this.weapon = "Mace";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

JavaScript doesn't care about case order, so we can also move around our numerical cases.

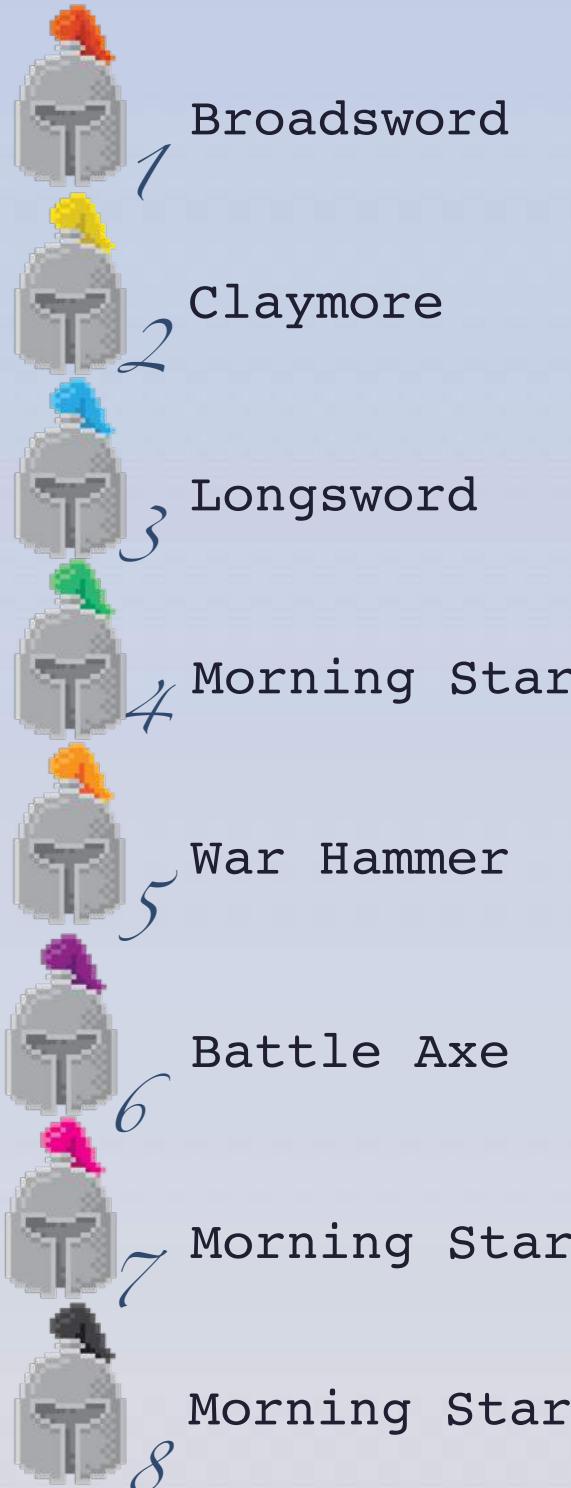


```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

JavaScript doesn't care about case order, so we can also move around our numerical cases.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 4:  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

JavaScript doesn't care about case order, so we can also move around our numerical cases.



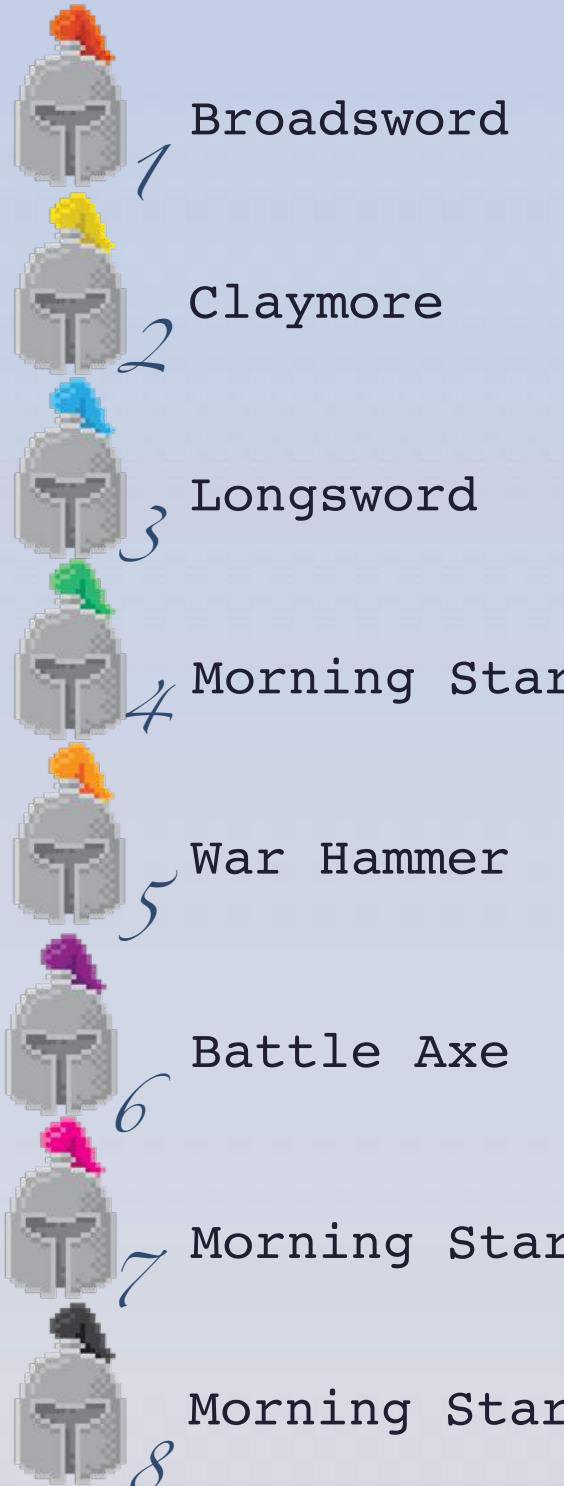
```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```

We can stack as many cases as we'd like on a set of actions.



WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

JavaScript doesn't care about case order, so we can also move around our numerical cases.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```

```
var soldier = new Knight("Richard", 4);  
console.log(soldier.weapon);
```

✓ → Morning Star

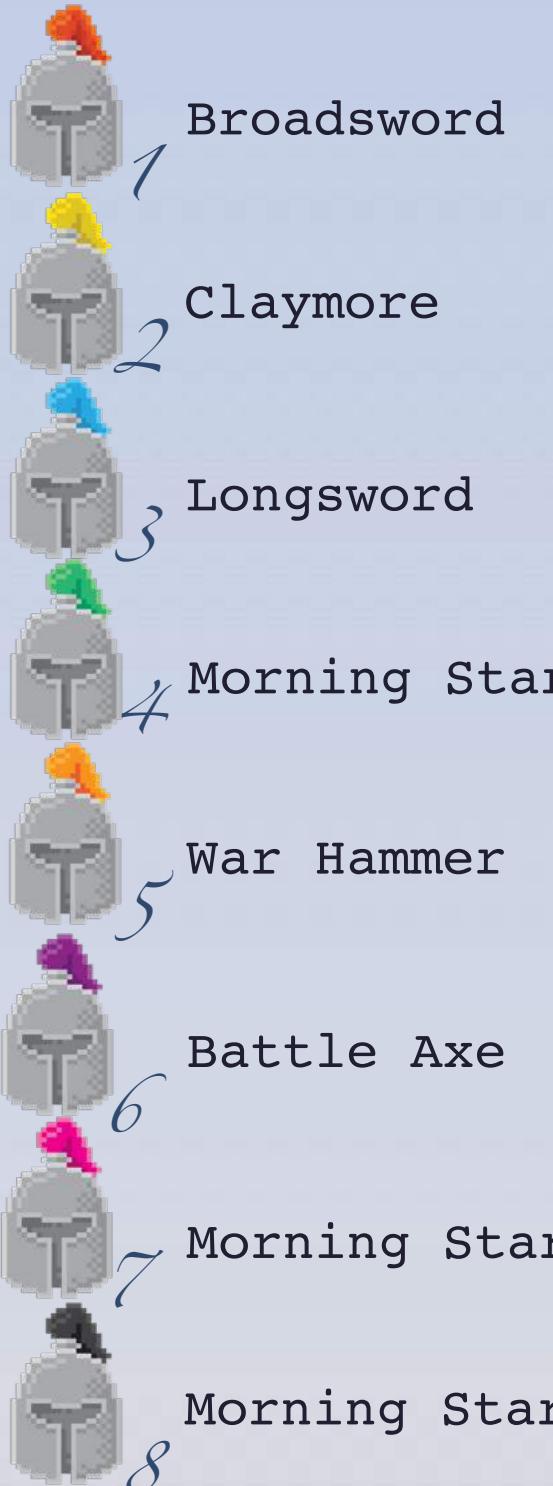
case 6:
 this.weapon = "Battle Axe";
 break;
case 4:
case 7:
case 8:
 this.weapon = "Morning Star";
 break;

Fall-through!



WHAT HAPPENS IF A CASE VALUE IS NOT PRESENT?

If a case value is not present in the switch block, no action will trigger.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
    }  
}
```

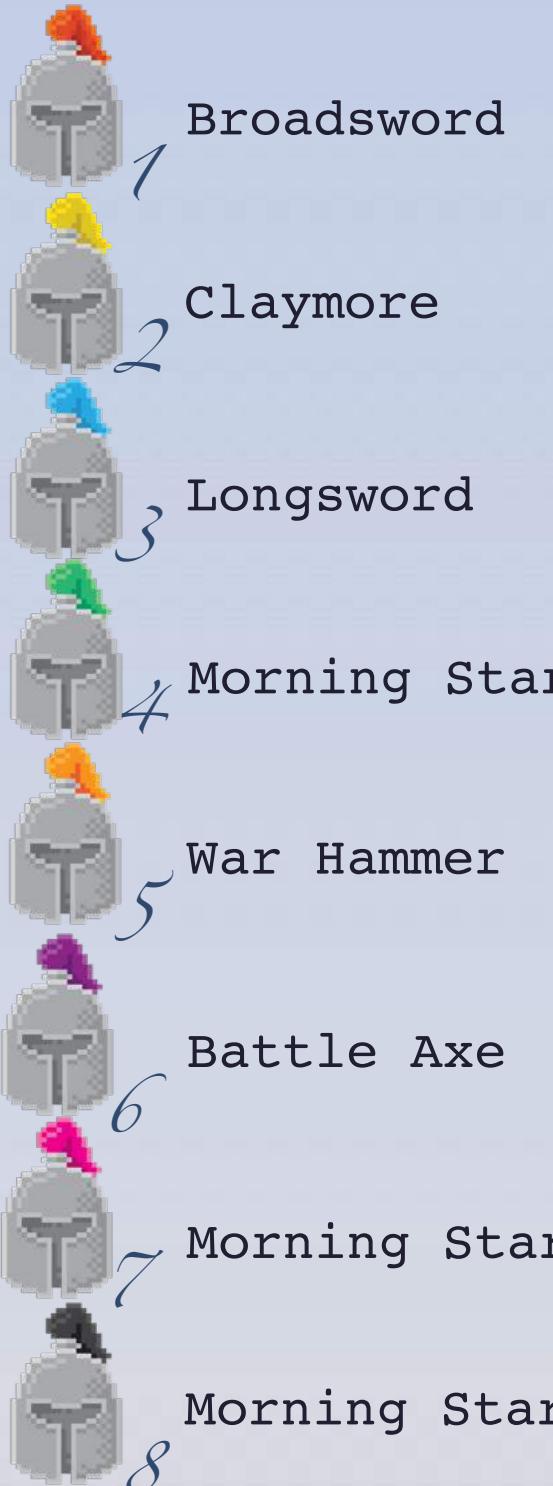
```
var king = new Knight("Arthur", "King");  
console.log(king.weapon);
```

→ undefined

The **weapon** property is never set to a value, but we can fix that...

LET'S MAKE SURE ARTHUR GETS HIS SWORD

Case values can be any JavaScript value that is “matchable”.



```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
        case "King":  
            this.weapon = "Excalibur";  
            break;  
    }  
}
```

A string is a perfectly acceptable case label.

```
var king = new Knight("Arthur", "King");  
console.log(king.weapon);
```



→ Excalibur

VERILY, A KNIGHT WITH NO WEAPON SUCKETH MUCH

The ‘default’ case can help us watch for unavailable regiment values and alert the armorer.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
        case "King":  
            this.weapon = "Excalibur";  
            break;  
    }  
}
```

VERILY, A KNIGHT WITH NO WEAPON SUCKETH MUCH

The ‘default’ case can help us watch for unavailable regiment values and alert the armorer.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
    }  
  
    var soldier3 = new Knight("Jerome", 12);
```

```
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
        case "King":  
            this.weapon = "Excalibur";  
            break;  
        default:  
            alert(name + " has an incorrect " +
```



The page at <https://www.codeschool.com>
says:

Jerome has an incorrect regiment, Master Armorer!

No weapon assigned!

OK

No match found? Use the **default**.

WHAT ABOUT THE FINAL BREAK STATEMENT?

Since no other case exists after our last one, we don't really need to force a switch break.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
        case "King":  
            this.weapon = "Excalibur";  
            break;  
        default:  
            alert(name + " has an incorrect " +  
                  "regiment, Master Armourer!" +  
                  "\n\nNo weapon assigned!");  
    }  
}
```

No **break** included in the default case,
since it's the last one anyway.

FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
  
        case "Knight": this.rubies = 6;  
    }  
}
```



FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```



FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```



FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```



FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
  
        case "King": this.diamonds = 1;  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```



FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
        case "King": this.diamonds = 1;  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```

FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
        case "King": this.diamonds = 1;  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```

Jumps straight to the correct case, with no fall-through before or after. Thus, no other gemstones.

```
var knightsDagger = new ceremonialDagger("Jerome", "Knight");
```

FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
        case "King": this.diamonds = 1;  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```

Jumps straight to the correct case, with no fall-through before or after. Thus, no other gemstones.

```
var knightsDagger = new ceremonialDagger("Jerome", "Knight");  
console.log(knightsDagger);
```

→ ceremonialDagger {length: 8, owner: "Jerome", rubies: 6}

FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
        case "King": this.diamonds = 1;  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```

Gets Field Marshal
gems, but then
experiences fall-
through, acquiring
lower ranked gems.

```
var marshalsDagger = new ceremonialDagger("Timothy", "Field Marshal");  
console.log(marshalsDagger);
```

→ ceremonialDagger {length: 8, owner: "Timothy",
sapphires: 4,
emeralds: 1,
rubies: 6}

FALL-THROUGH ALLOWS HIERARCHICAL CODE EXECUTION

A carefully organized switch block can add LEAST common properties first and MOST common, last.



```
function ceremonialDagger(knight, rank){  
    this.length = 8;  
    this.owner = knight;  
    switch(rank){  
        case "King": this.diamonds = 1; ←  
        case "High Constable": this.amethyst = 2;  
        case "Field Marshal": this.sapphires = 4;  
        case "Captain": this.emeralds = 1;  
        case "Knight": this.rubies = 6;  
    }  
}
```

Arthur gets his diamond, but then experiences fall-through, acquiring all of the lower ranked gems.

```
var kingsDagger = new ceremonialDagger("Arthur", "King");  
console.log(kingsDagger);
```

→ ceremonialDagger {length: 8, owner: "Arthur", diamonds: 1, amethyst: 2, sapphires: 4, emeralds: 1, rubies: 6}

THE PENDANT OF PERFORMANCE

Section 1
Loop Optimization

JAVASCRIPT
BEST PRACTICES

A COMMON FOR-LOOP SCENARIO

Memory access is not necessarily ideal in our normal for-loop structure.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

- You've found the following necklaces:
- ruby
- pearl
- sapphire
- diamond

treasureChest

```
goldCoins: 10,000,  
magicalItem: "Crown of Speed",  
necklaces: ["ruby", "pearl",  
           "sapphire",  
           "diamond"],  
openLid: function () {  
  alert("Creeeeeak!");  
}
```



A COMMON FOR-LOOP SCENARIO

Memory access is not necessarily ideal in our normal for-loop structure.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

At the start of each potential loop cycle,
the program will need to find and retrieve:

1. the value of `i`
2. the `treasureChest` object
3. the `necklaces` property
4. the array pointed to by the property
5. the `length` property of the array



WHAT STEPS CAN WE ELIMINATE?

Any intermediary step to the bit of data we need is a candidate for death.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

...so we don't really need
all these other steps.



```
treasureChest
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
length: 4
openLid: function () {
  alert("Creeeee!");
```

A hand-drawn style diagram of a treasure chest filled with gold coins. To the right of the chest is a dark rectangular box containing code. A purple arrow points from the text "...so we don't really need all these other steps." to the word "length". Another purple arrow points from the same text to the "length" variable in the code. A pink arrow points from the "length" variable in the code to the "length" variable in the JSON object. A yellow oval highlights the "length" variable in the JSON object. A pink arrow also points from the "length" variable in the JSON object to the "length" variable in the code.

All we are interested in during loop control
is this single value and the loop counter.

LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
for(var i = 0; i < treasureChest.necklaces.length; i++) {
  console.log(treasureChest.necklaces[i]);
}
```



```
treasureChest
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
length: 4
openLid: function () {
  alert("Creeeeak!");
}
```

LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x =
for(var i = 0; i < treasureChest.necklaces.length; i++) {
  console.log(treasureChest.necklaces[i]);
}
```



```
treasureChest
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
length: 4
openLid: function () {
  alert("Creeeeak!");
}
```

LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i <
    console.log(treasureChest.necklaces[i]);
}
```

First, we assign the `length`'s value to a new variable. This will use all of the accesses as before, but we'll only do it once.



LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i <
    console.log(treasureChest.necklaces[i]);
}
```



LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x;
    console.log(treasureChest.necklaces[i]);
}
```



```
treasureChest →
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"], →
length: 4
openLid: function () {
    alert("Creeeeeak!");
}
```

LOCAL VARIABLES PROVIDE SPEED

We can use “cached values” to curtail lengthy, repetitive access to the same data.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```



Now we'll let our loop use the local `x` during control. The majority of extraneous steps have now been eliminated.



PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

Memory access during loop control now only needs to:

1. retrieve the value of `i`
2. retrieve the value of `x`

Then, add in our one-time cost in creating `x`:

1. creating the variable `x` in memory.
- 2-5. the 4 steps finding the value of `length`

```
treasureChest
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

- 2** steps x (**4** executed loops + **1** check to stop)
- = **10** steps for the processor, just in memory access.
- + **5** steps in the creation of the extra variable.

- = **15** steps of strictly memory access to execute the loop

treasureChest

```
goldCoins: 10,000,  
magicalItem: "Crown of Speed",  
necklaces: ["ruby", "pearl",  
           "sapphire",  
           "diamond"],  
openLid: function () {  
  alert("Creeeeeak!");  
}
```



PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeak!");
}
```



15 steps of strictly memory access to execute the loop

PROCESSOR SAVINGS MEANS MORE SPEED

Let's check out what we've saved on our relatively simple example.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

25 steps in our previous version

- **15** steps of strictly memory access to execute the loop



10 steps of savings

treasureChest

goldCoins: 10,000,

magicalItem: "Crown of Speed",

necklaces: ["ruby", "pearl",

"sapphire",
"diamond"],

openLid: function () {
 alert("Creeeeeak!");
}



WHAT IF WE HAD 10,000 NECKLACES?

With large data amounts, this improvement produces really noticeable results.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

5 steps x (10,000 executed loops + 1 check to stop)

= 50,005 memory access steps

2 steps x (10,000 executed loops + 1 check to stop)

= 20,002 memory access steps

+ 5 extra steps for x

20,007 memory access steps

treasureChest

goldCoins: 10,000,

magicalItem: "Crown of Speed",

necklaces: ["ruby", "pearl",
"sapphire",
"diamond"],

```
openLid: function () {
  alert("Creeeeeak!");
}
```



WHAT IF WE HAD 10,000 NECKLACES?

With large data amounts, this improvement produces really noticeable results.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

50,005 memory access steps
- **20,002** memory access steps



~30,000 steps of savings!

treasureChest

```
goldCoins: 10,000,  
magicalItem: "Crown of Speed",  
necklaces: ["ruby", "pearl",  
           "sapphire",  
           "diamond"],  
openLid: function () {  
  alert("Creeeeeak!");  
}
```



PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest



```
goldCoins: 10,000,  
magicalItem: "Crown of Speed",  
necklaces: ["ruby", "pearl",  
           "sapphire",  
           "diamond"],  
  
openLid: function () {  
  alert("Creeeeak!");  
}
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
var x = treasureChest.necklaces.length;
for(var i = 0,
    console.log(treasureChest.necklaces[i]);
}
```



Surprise! A comma will allow us to execute multiple statements within the first parameter of the loop.

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
    var x = treasureChest.necklaces.length;
for(var i = 0,                                i < x; i++) {
    console.log(treasureChest.necklaces[i]);
}
```

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
    alert("Creeeeeak!");
}
```



PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");

for(var i = 0, var x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, var x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```



This `var` is now unnecessary, because the comma tells the compiler that more variables will be declared ... and maybe even assigned.

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0,      x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

By creating both variables in the loop parameters, we signal they're only intended for use inside the loop.

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

PLACE YOUR EXTRA CONTROL VARIABLES INSIDE THE LOOP

For better organization, we can place our new control variable inside the first loop parameter.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

```
console.log(x);
```



→ 4

Beware! JavaScript does not scope to blocks, so any variables declared in loops will be available after the loop, and may overwrite pre-existing globals!

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

Here's another access of a property within an object.



```
treasureChest
  goldCoins: 10,000,
  magicalItem: "Crown of Speed",
  necklaces: ["ruby", "pearl",
    "sapphire",
    "diamond"],
  openLid: function () {
    alert("Creeeeeak!");
  }
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(treasureChest.necklaces[i]);
}
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(
    list[i]);
}
```

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
  "sapphire",
  "diamond"],
openLid: function () {
  alert("Creeeeeak!");}
```



MORAL: AVOID REPETITIVE ACCESS AT DEPTH

We can even find another place to further streamline this process, maximizing speed.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(list[i]);
}
```



Now we've avoided the extra step of accessing the `treasureChest` in each cycle.

`treasureChest`

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(list[i]);
}
```

This function would now be available to all Arrays in our program.

```
Array.prototype.countType = function (type) {
  var count = 0;
  for(var i = 0, x = this.length; i < x; i++) {
    if(this[i] === type) {
      count++;
    }
  }
  return count;
};
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(list[i]);
}
```



treasureChest

```
Array.prototype.countType = function (type) {
};

};
```

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(list[i]);
}
```

```
Array.prototype.removeAll = function (item) {
  var removed = [ ];
  for(var i = 0, x = this.length; i < x; i++){
    if(this[i] === item){
      removed.push(item);
      this.splice(i, 1);
    }
  }
  return removed;
};
```

```
Array.prototype.countType = function (type) {
};
...;
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
    console.log(list[i]);
}
```

```
Array.prototype.countType = function (type) {  
    ...  
};
```

treasureChest



```
goldCoins: 10,000,  
magicalItem: "Crown of Speed",  
necklaces: [ "ruby", "pearl",  
             "sapphire",  
             "diamond" ],  
openLid: function () {  
    alert("Creeeeeak!");  
}
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(list[i]);
}
```



treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

```
Array.prototype.removeAll = function (item) {
  ...
};

Array.prototype.countType = function (type) {
  ...
};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(var i = 0, x = treasureChest.necklaces.length; i < x; i++) {
  console.log(list[i]);
}
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```

```
Array.prototype.removeAll = function (item) {
  ...
};

Array.prototype.countType = function (type) {
  ...
};
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(p in list
    ) {
  console.log(list[i]);
}
```

```
Array.prototype.removeAll = function (item) {
  ...
};

Array.prototype.countType = function (type) {
  ...
};
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");}
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(p in list) {
  console.log(list[i]);
}
```

```
Array.prototype.removeAll = function (item) {
  ...
};

Array.prototype.countType = function (type) {
  ...
};
```

treasureChest



```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
            "sapphire",
            "diamond"],
openLid: function () {
  alert("Creeeeeak!");}
```

CHOOSING THE BEST LOOP FOR ARRAYS

Stick with for-loops instead of for-in loops when your goal is only to reach every index.

```
console.log("You've found the following necklaces:");
var list = treasureChest.necklaces;
for(p in list) {
  console.log(list[i]);
}
```

- You've found the following necklaces:
- ruby
- pearl
- sapphire
- diamond
- removeAll
- countType



Using a property approach to access indices will also add in ALL methods that have been added to the Array prototype.

```
Array.prototype.removeAll = function (item) {
  ...
};

Array.prototype.countType = function (type) {
  ...
};
```

Methods we add to the prototype become "enumerable" just like indices.

treasureChest

```
goldCoins: 10,000,
magicalItem: "Crown of Speed",
necklaces: ["ruby", "pearl",
  "sapphire",
  "diamond"],
openLid: function () {
  alert("Creeeeeak!");
}
```



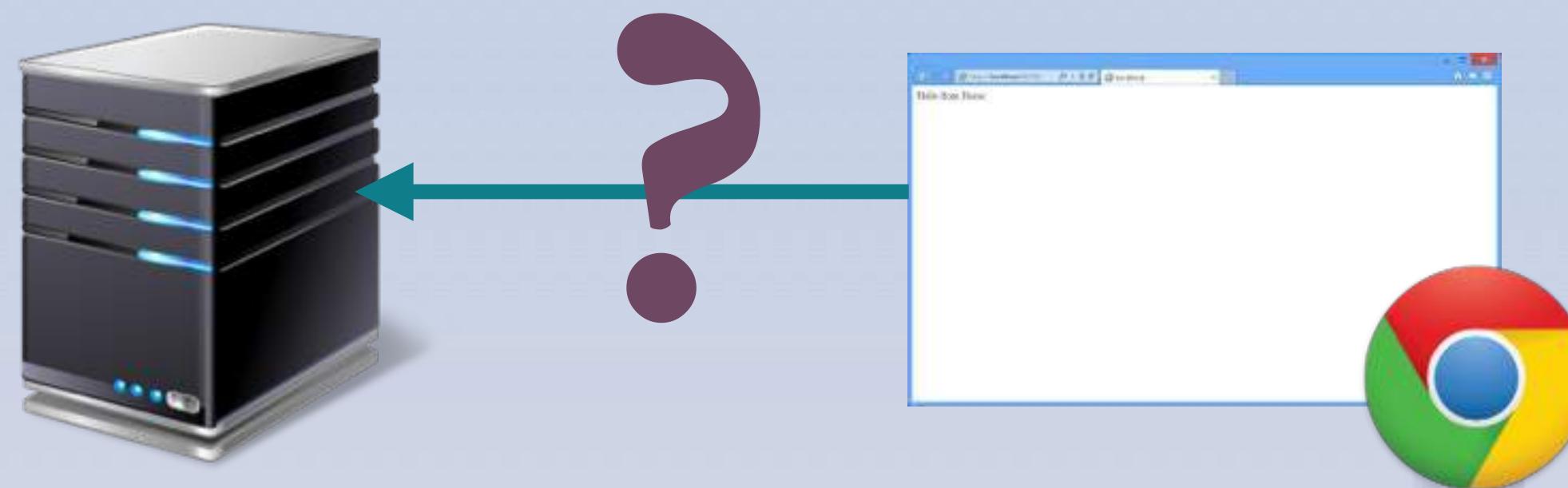
THE PENDANT OF PERFORMANCE

Section 2
Script Execution

JAVASCRIPT
BEST PRACTICES

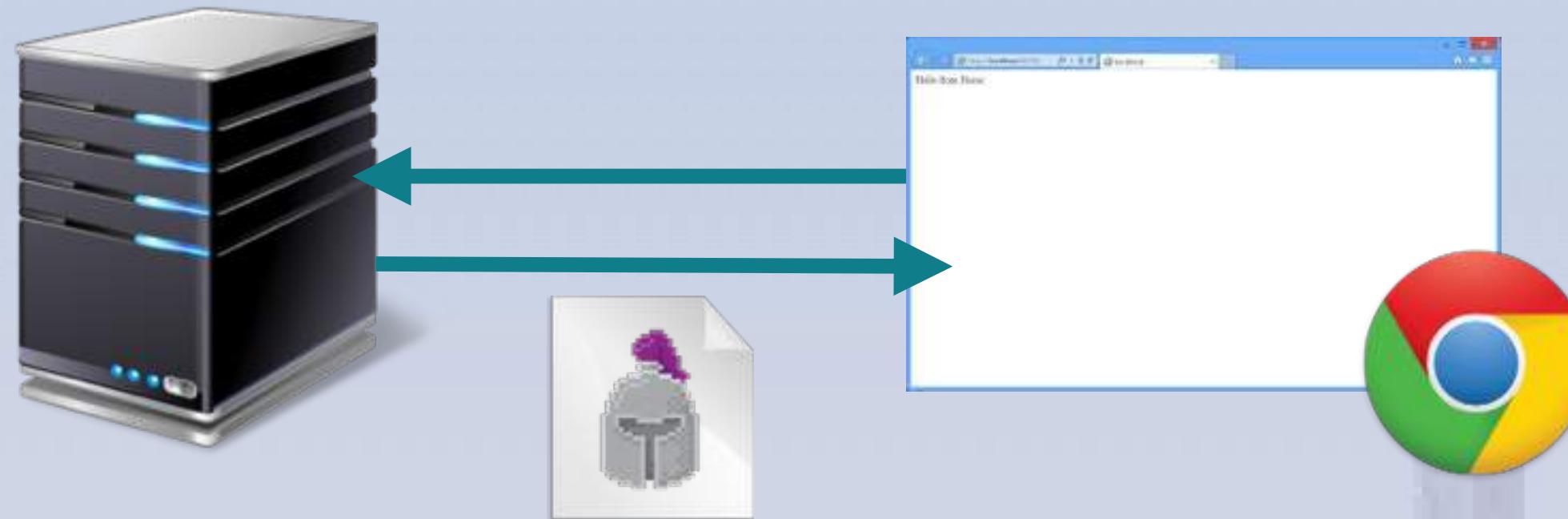
SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



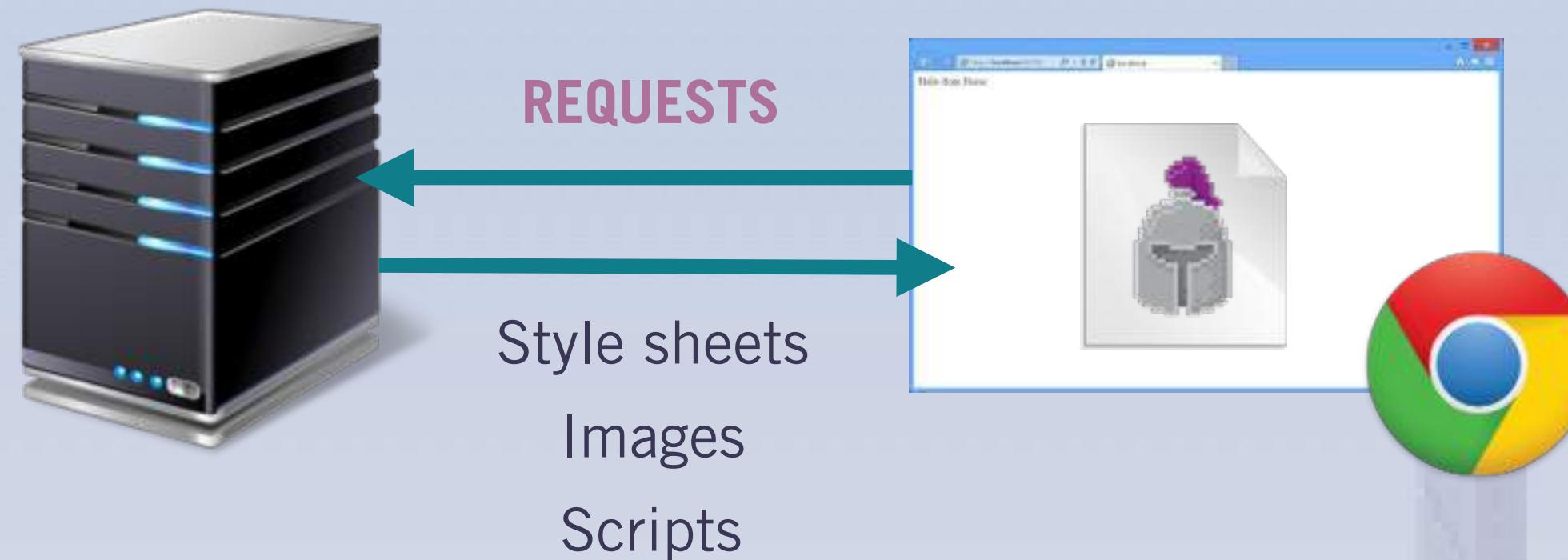
SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



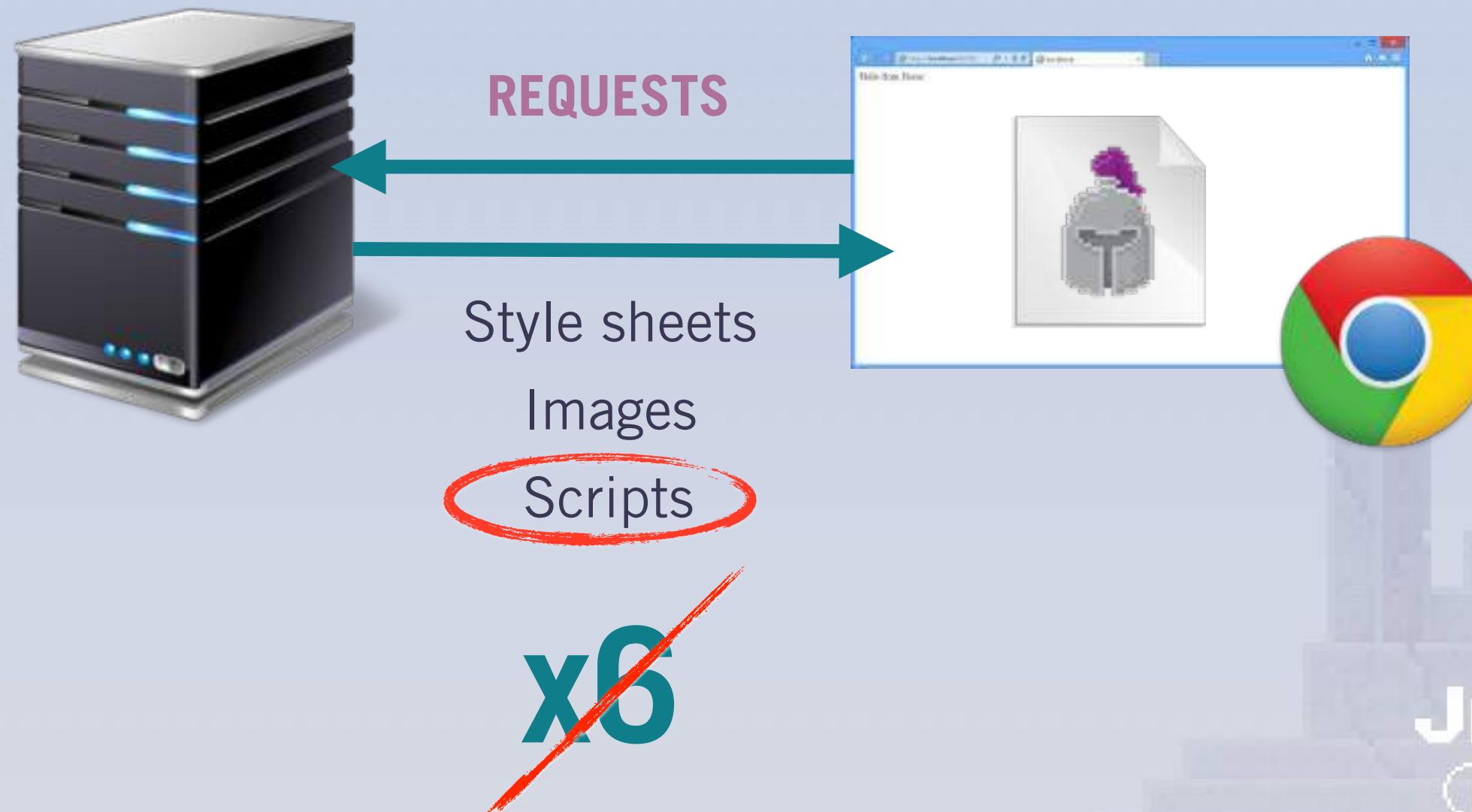
SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



SCRIPT PLACEMENT IMPACTS PERFORMANCE

Let's take a closer look at how a browser retrieves and acts on scripts.



JAVASCRIPT
BEST PRACTICES

WHERE WOULD WE FIND TROUBLING SCRIPTS?

Scripts encountered high in the <head> or <body> tags of an HTML page can have adverse effects.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
           src="http://www.ThirdReg.com/sparring.js"></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>

    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



This script tag seems harmless enough, right? Just a quick download of a file, and then we'll get on with our lives ...

WHERE WOULD WE FIND TROUBLING SCRIPTS?

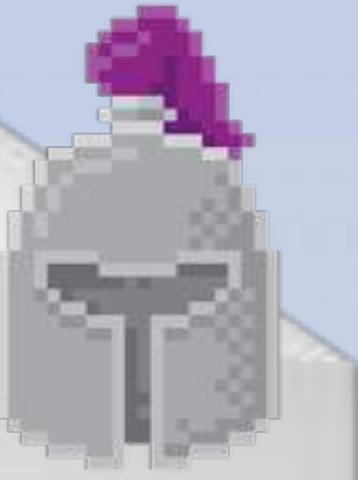
Scripts encountered high in the <head> or <body> tags of an HTML page can have adverse effects.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>

    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



SPARRING.JS

- builds a huge array of soldiers from a separate file.
- calculates all the possible unique groups for:
 - sparring groups of 2
 - sparring groups of 3
 - sparring groups of 4
- randomizes order of those groups
- collects groups in a pairs for larger matches
- ...more processes...
- ...more processes...
- ...more processes...

...tick-tick-tick-tick-tick-tick...



ONE SOLUTION: RELOCATE WORK-INTENSIVE SCRIPTS

Scripts that are not essential to immediate loading of the page should be moved as low as possible.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>

    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



SPARRING.JS

- builds a huge array of soldiers from a separate file.
- calculates all the possible unique groups for:
 - sparring groups of 2
 - sparring groups of 3
 - sparring groups of 4
- randomizes order of those groups
- collects groups in a pairs for larger matches
- ...more processes...
- ...more processes...
- ...more processes...



...tick-tick-tick-tick-tick-tick...

ONE SOLUTION: RELOCATE WORK-INTENSIVE SCRIPTS

Scripts that are not essential to immediate loading of the page should be moved as low as possible.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>

    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



ONE SOLUTION: RELOCATE WORK-INTENSIVE SCRIPTS

Scripts that are not essential to immediate loading of the page should be moved as low as possible.



```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>
    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>
    (...and perhaps a bunch of other important HTML...)
    <script type="text/javascript"
           src="http://www.ThirdReg.com/sparring.js"></script>
  </body>
</html>
```

Now, most of the visual and informational components of the site will become available to the user before the script is loaded and processed.

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 `async` attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>
    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>
    (...and perhaps a bunch of other important HTML...)
    <script type="text/javascript"
           src="http://www.ThirdReg.com/sparring.js"></script>
  </body>
</html>
```



ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 `async` attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>

    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



If we add an `async` attribute inside our `script` tag, we'll prevent the script from blocking the page's load.

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 `async` attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"
      ></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>
    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 `async` attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>

    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"
      async></script>

    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>
    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



Though we've got it in the back here, the `async` attribute could be added anywhere inside our tag.

ANOTHER SOLUTION: RUNNING SCRIPTS ASYNCHRONOUSLY

With external files, the HTML5 `async` attribute will allow the rest of the page to load before the script runs.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>The Third Regiment</h1>
    <script type="text/javascript"
      src="http://www.ThirdReg.com/sparring.js"
      async></script>
    <p>
      Sparring event at 3:30 Tuesday, homies!
      
    </p>
    (...and perhaps a bunch of other important HTML...)
  </body>
</html>
```



The browser will now fetch the script resource, but will continue parsing and rendering HTML, without waiting for the script to complete!

PERFORMANCE

Section 2
Script Execution

JAVASCRIPT
BEST PRACTICES

THE PENDANT OF PERFORMANCE

Section 3
Short Performance Tips

JAVASCRIPT
BEST PRACTICES

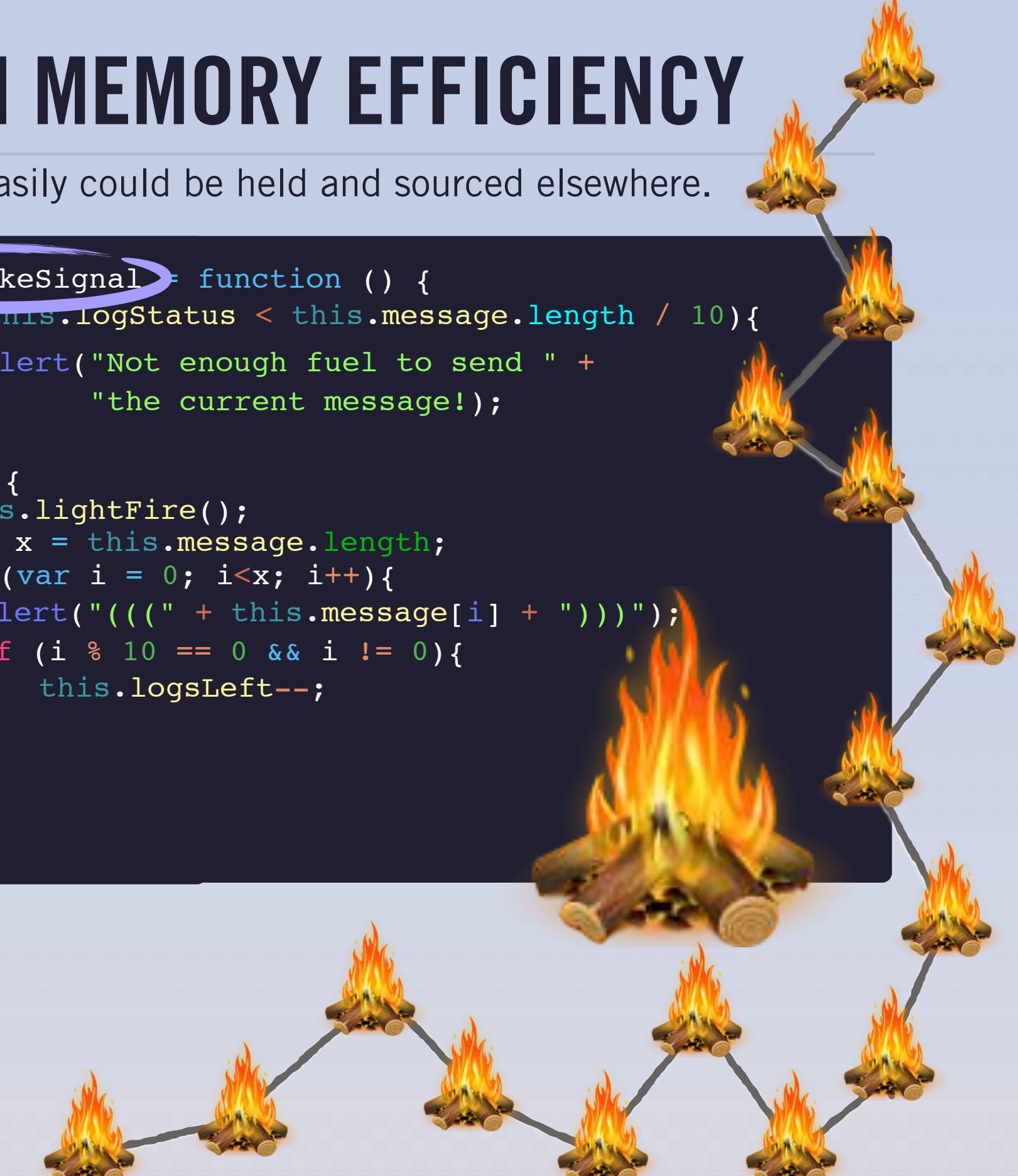
LET INHERITANCE HELP WITH MEMORY EFFICIENCY

Beware of loading up individual objects with code that easily could be held and sourced elsewhere.

```
function SignalFire( ID, startingLogs ){  
    this.fireID = ID;  
    this.logsLeft = startingLogs;  
  
    this.addLogs = function ( numLogs ){  
        this.logsLeft += numLogs;  
    }  
    this.lightFire = function () {  
        alert("whoooosh!");  
    }  
}  
  
this.smokeSignal = function () {  
    if (this.logStatus < this.message.length / 10){  
        alert("Not enough fuel to send " +  
              "the current message!");  
    } else {  
        this.lightFire();  
        var x = this.message.length;  
        for(var i = 0; i < x; i++){  
            alert("(((( " + this.message[i] + " ))))");  
            if (i % 10 == 0 && i != 0){  
                this.logsLeft--;  
            }  
        }  
    }  
}
```



We don't need to build all of these methods within every single `SignalFire` object, which would use extra memory AND take longer to create.



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){  
    this.fireID = ID;  
    this.logsLeft = startingLogs;  
  
    this.addLogs = function ( numLogs ){  
        this.logsLeft += numLogs;  
    }  
    this.lightFire = function () {  
        alert("Whoooosh!");  
    }  
}  
  
this.smokeSignal = function () {  
    if (this.logStatus < this.message.length / 10){  
        alert("Not enough fuel to send " +  
              "the current message!");  
    }  
    else {  
        this.lightFire();  
        var x = this.message.length;  
        for(var i = 0; i<x; i++){  
            alert("(((( " + this.message[i] + " ))))");  
            if (i % 10 == 0 && i != 0){  
                this.logsLeft--;  
            }  
        }  
    }  
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ) {
    this.fireID = ID;
    this.logsLeft = startingLogs;

    this.addLogs = function ( numLogs ){
        this.logsLeft += numLogs;
    }

    this.lightFire = function () {
        alert("Whoooosh!");
    }
}

this.smokeSignal = function () {
    if (this.logStatus < this.message.length / 10){
        alert("Not enough fuel to send " +
              "the current message!");
    }
    else {
        this.lightFire();
        var x = this.message.length;
        for(var i = 0; i<x; i++){
            alert(((( " + this.message[i] + " ))));
            if (i % 10 == 0 && i != 0){
                this.logsLeft--;
            }
        }
    }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ) {
    this.fireID = ID;
    this.logsLeft = startingLogs;

    this.addLogs = function ( numLogs ){
        this.logsLeft += numLogs;
    }

    this.lightFire = function () {
        alert("Whoooosh!");
    }

    this.smokeSignal = function () {
        if (this.logStatus < this.message.length / 10){
            alert("Not enough fuel to send " +
                  "the current message!");
        }
        else {
            this.lightFire();
            var x = this.message.length;
            for(var i = 0; i<x; i++){
                alert(((( " + this.message[i] + " ))));
                if (i % 10 == 0 && i != 0){
                    this.logsLeft--;
                }
            }
        }
    }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ) {
    this.fireID = ID;
    this.logsLeft = startingLogs;

    addLogs   function ( numLogs ){
        this.logsLeft += numLogs;
    }

    lightFire  function () {
        alert("Whoooosh!");
    }

    smokeSignal  function () {
        if (this.logStatus < this.message.length / 10){
            alert("Not enough fuel to send " +
                  "the current message!");
        }
        else {
            this.lightFire();
            var x = this.message.length;
            for(var i = 0; i<x; i++){
                alert(((( " + this.message[i] + " ))));
                if (i % 10 == 0 && i != 0){
                    this.logsLeft--;
                }
            }
        }
    }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ) {
    this.fireID = ID;
    this.logsLeft = startingLogs;

    addLogs: function ( numLogs ){
        this.logsLeft += numLogs;
    }

    lightFire: function () {
        alert("Whoooosh!");
    }

    smokeSignal: function () {
        if (this.logStatus < this.message.length / 10){
            alert("Not enough fuel to send " +
                  "the current message!");
        }
        else {
            this.lightFire();
            var x = this.message.length;
            for(var i = 0; i<x; i++){
                alert(((( " + this.message[i] + " ))));
                if (i % 10 == 0 && i != 0){
                    this.logsLeft--;
                }
            }
        }
    }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){  
    this.fireID = ID;  
    this.logsLeft = startingLogs;  
  
    SignalFire.prototype = {  
        addLogs: function ( numLogs ){  
            this.logsLeft += numLogs;  
        }  
        lightFire: function () {  
            alert("Whoooosh!");  
        }  
        smokeSignal: function () {  
            if (this.logStatus < this.message.length / 10){  
                alert("Not enough fuel to send " +  
                    "the current message!");  
            }  
            else {  
                this.lightFire();  
                var x = this.message.length;  
                for(var i = 0; i<x; i++){  
                    alert(((( " + this.message[i] + " ))));  
                    if (i % 10 == 0 && i != 0){  
                        this.logsLeft--;  
                    }  
                }  
            }  
        }  
    }  
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
    this.fireID = ID;
    this.logsLeft = startingLogs;
}
```

```
SignalFire.prototype = {
    addLogs: function ( numLogs ) {
        this.logsLeft += numLogs;
    }
    lightFire: function () {
        alert("Whoooosh!");
    }
    smokeSignal: function () {
        if (this.logStatus < this.message.length / 10) {
            alert("Not enough fuel to send " +
                "the current message!");
        } else {
            this.lightFire();
            var x = this.message.length;
            for(var i = 0; i < x; i++) {
                alert("(((( " + this.message[i] + " ))))");
                if (i % 10 == 0 && i != 0) {
                    this.logsLeft--;
                }
            }
        }
    }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){
  this.fireID = ID;
  this.logsLeft = startingLogs;
}
```

```
SignalFire.prototype = {
  addLogs: function ( numLogs ) {
    this.logsLeft += numLogs;
  }
  lightFire: function () {
    alert("Whoooosh!");
  }
  smokeSignal: function () {
    if (this.logStatus < this.message.length / 10) {
      alert("Not enough fuel to send " +
        "the current message!");
    } else {
      this.lightFire();
      var x = this.message.length;
      for(var i = 0; i < x; i++) {
        alert("(((( " + this.message[i] + " ))))");
        if (i % 10 == 0 && i != 0) {
          this.logsLeft--;
        }
      }
    }
  }
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){  
    this.fireID = ID;  
    this.logsLeft = startingLogs;  
}
```

```
var fireOne = new SignalFire(1, 20);  
var fireTwo = new SignalFire(2, 15);  
var fireThree = new SignalFire(3, 24);
```

```
fireOne.addLogs(8);  
fireTwo.addLogs(10);  
fireThree.addLogs(4);
```

Now, the `addLogs` method is found in one useful place, instead of being replicated across all `signalFires`.

```
SignalFire.prototype = {  
    addLogs: function ( numLogs ){  
        this.logsLeft += numLogs;  
    },  
    lightFire: function () {  
        alert("Whoooosh!");  
    },  
    smokeSignal: function () {  
        if (this.logStatus < this.message.length / 10){  
            alert("Not enough fuel to send " +  
                  "the current message!");  
        }  
        else {  
            this.lightFire();  
            var x = this.message.length;  
            for(var i = 0; i < x; i++){  
                alert("((( " + this.message[i] + " ))));  
                if (i % 10 == 0 && i != 0){  
                    this.logsLeft--;  
                }  
            }  
        }  
    }  
}
```



USE A PROTOTYPE FOR SHARED STUFF

Give all common methods that a “class” of objects will use to the constructor’s prototype.

```
function SignalFire( ID, startingLogs ){  
    this.fireID = ID;  
    this.logsLeft = startingLogs;  
}
```

```
var fireOne = new SignalFire(1, 20);  
var fireTwo = new SignalFire(2, 18);  
var fireThree = new SignalFire(3, 24);
```

```
fireOne.addLogs(8);  
fireTwo.addLogs(10);  
fireThree.addLogs(4);
```

```
fireThree.smokeSignal("Goblins!");
```

```
SignalFire.prototype = {  
    addLogs: function ( numLogs ){  
        this.logsLeft += numLogs;  
    }  
    lightFire: function () {  
        alert("Whoo-hoo!");  
    }  
    smokeSignal:  
        if (this.logsLeft <= 0){  
            alert("(((!)))");  
        }  
        else {  
            this.lightFire();  
            var x = this.message.length;  
            for(var i = 0; i < x; i++){  
                alert("((( " + this.message[i] + " ))));  
                if (i % 10 == 0 && i != 0){  
                    this.logsLeft--;  
                }  
            }  
        }  
    }  
}
```



The page at <https://www.codeschool.com>
says:

OK



ADDING INDIVIDUAL DOM ELEMENTS AIN'T ALWAYS SPEEDY

Each new addition to the DOM causes document “reflow”, which can really hinder user experience.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];

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



Each time the list is appended, we access the DOM and cause an entire document reflow. Not as speedy as we'd like, especially if our list was huge...

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];

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

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
        src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
    var element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    list.appendChild(element);
}
```

First we create a fragment, which will function as a staging area to hold all of our new li elements.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
    var element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    .appendChild(element);
}
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
        src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```

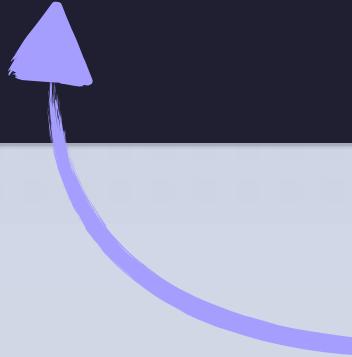


USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i < x; i++) {
  var element = document.createElement("li");
  element.appendChild( document.createTextNode( kotw[i] ) );
  fragment.appendChild(element);
}
```



Now we add each new `li` element to the staging fragment, instead of to the document itself.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



USE A DOCUMENT FRAGMENT TO INSERT ADDITIONS ALL AT ONCE

Fragments are invisible containers that hold multiple DOM elements without being a node itself.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
    var element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```



Finally, we append all of our new text nodes in one fell swoop, using only one DOM touch!

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



BONUS BP: DECLARE VARIABLES AS FEW TIMES AS POSSIBLE

Every `var` keyword adds a look-up for the JavaScript parser that can be avoided with comma extensions.

KOTW.JS

```
var list = document.getElementById("kotwList");
var kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"];
var fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
    var element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
        src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



BONUS BP: DECLARE VARIABLES AS FEW TIMES AS POSSIBLE

Every `var` keyword adds a look-up for the JavaScript parser that can be avoided with comma extensions.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
    var element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```

Commas used after an initial declaration can signal that you'll be declaring further variables. It's also arguably more legible for those reading your code.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment();
for (var i = 0, x = kotw.length; i<x; i++) {
    var element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```

This variable will be declared
every time the loop executes!

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment()
for (var i = 0, x = kotw.length; i<x; i++) {
    element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment()
for (var i = 0, x = kotw.length; i<x; i++) {
    element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



DECLARING IN LOOPS SHOULD THUS BE USED WITH CAUTION

Anticipate variable needs to avoid the processor burden of creating a new var over and over.

KOTW.JS

```
var list = document.getElementById("kotwList"),
    kotw = ["Jenna Rangespike",
            "Neric Farthing",
            "Darkin Stonefield"],
    fragment = document.createDocumentFragment(),
    element;
for (var i = 0, x = kotw.length; i < x; i++) {
    element = document.createElement("li");
    element.appendChild( document.createTextNode( kotw[i] ) );
    fragment.appendChild(element);
}
list.appendChild(fragment);
```

In the tradeoff, we've opted to avoid processor burden instead of having `element` declared where it is used.

```
<!DOCTYPE html>
<html>
<body>

<h1>Knights of the Week!</h1>
<ul id="kotwList"></ul>

**some more HTML here**

<script type="text/javascript"
       src="http://www.ThirdReg.com/kotw.js">
</script>

</body>
</html>
```



EFFICIENT CHOICES FOR STRING CONCATENATION

In building strings, different methods will yield different results in terms of execution speed.

```
var knight = "Jenna Rangespike";
var action = " strikes the dragon with a ";
var weapon = "Halberd";
```

```
var turn = "";
turn += knight;
turn += action;
turn += weapon;
```



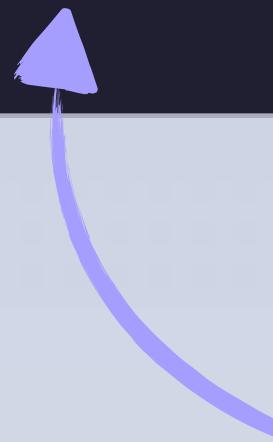
The standard concatenation operator has been optimized in most modern browser versions, and is an ideal choice for a small number of string concatenations.

EFFICIENT CHOICES FOR STRING CONCATENATION

In building strings, different methods will yield different results in terms of execution speed.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
    ***a hundred or more other html elements***,
    "</script>", "</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```



While concatenation does get the job done, another method enjoys a performance boost when your strings are present in a array.

DECIDE WHAT'S BETTER BEFORE REACHING FOR **+ =**

For concatenations over an array's contents, use the **join()** method inherited from the Array prototype.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
    ***a hundred or more other html elements***,
    "</script>", "</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

```
page = newPageBuild.join("\n");
```



The **join** method concatenates each index of the array, “joined” by any string parameter you pass in. In addition to being faster in tests than many String methods, it is also easier to read.

DECIDE WHAT'S BETTER BEFORE REACHING FOR **+ =**

For concatenations over an array's contents, use the **join()** method inherited from the Array prototype.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
    ***a hundred or more other html elements***,
    "</script>", "</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

```
page = newPageBuild.join("\n");
```

DECIDE WHAT'S BETTER BEFORE REACHING FOR **+ =**

For concatenations over an array's contents, use the **join()** method inherited from the Array prototype.

```
var newPageBuild = [ "<!DOCTYPE html>", "<html>", "<body>", "<h1>",
    ***a hundred or more other html elements***,
    "</script>", "</body>", "</html>" ];
```

```
var page = "";
for(var i = 0, x = newPageBuild.length; i < x; i++){
    page += newPageBuild[i];
}
```

```
page = newPageBuild.join("\n");
console.log(page);
```

→ <!DOCTYPE html>
<html>
<body>
<h1>
...
</script>
</body>
</html>

THE PENDANT OF PERFORMANCE

Section 4
Measuring Performance I:

`Console.time`

JAVASCRIPT
BEST PRACTICES

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
function Knight (name, regiment){  
    this.name = name;  
    this.regiment = regiment;  
    switch (regiment) {  
        case 1:  
            this.weapon = "Broadsword";  
            break;  
        case 2:  
            this.weapon = "Claymore";  
            break;  
        case 3:  
            this.weapon = "Longsword";  
            break;  
        case 5:  
            this.weapon = "War Hammer";  
            break;  
        case 6:  
            this.weapon = "Battle Axe";  
            break;  
        case 4:  
        case 7:  
        case 8:  
            this.weapon = "Morning Star";  
            break;  
        case "King":  
            this.weapon = "Excalibur";  
            break;  
        default:  
            alert(name + " has an incorrect " +  
                  "regiment, Master Armourer!" +  
                  "\n\nNo weapon assigned!");  
    }  
}
```

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
```

```
function Knight (name, regiment){ 
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```



We'll steadily add each newbie Knight to the regiment's list of members.

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
    var newGuy = new Knight( firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
}
```

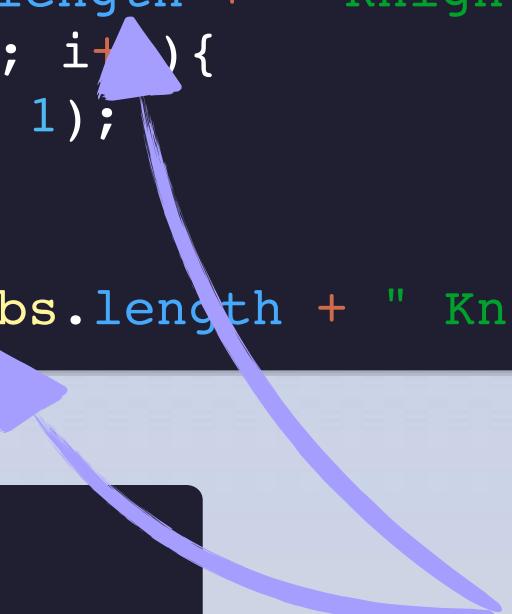
```
function Knight (name, regiment){ 
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
        case 1:
            this.weapon = "Broadsword";
            break;
        case 2:
            this.weapon = "Claymore";
            break;
        case 3:
            this.weapon = "Longsword";
            break;
        case 5:
            this.weapon = "War Hammer";
            break;
        case 6:
            this.weapon = "Battle Axe";
            break;
        case 4:
        case 7:
        case 8:
            this.weapon = "Morning Star";
            break;
        case "King":
            this.weapon = "Excalibur";
            break;
        default:
            alert(name + " has an incorrect " +
                  "regiment, Master Armourer!" +
                  "\n\nNo weapon assigned!");
    }
}
```

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i < x; i++) {
  var newGuy = new Knight( firstRegimentNewbs[i], 1 );
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment) {
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```



To unite timer boundaries into one timer, their parameter labels must match.

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i < x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1 );
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

→ Time to add 3 Knights: 0.036ms

```
function Knight (name, regiment){
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```

console.time automatically
prefaces the time measurement
with the label we passed in as a
parameter, plus a colon.

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){ 
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```

WANT TO TEST THE SPEED OF YOUR CODE?

Let's first look at a simple console method that will assess the time your code takes to run.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){ 
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```

→ Time to add 4 Knights: 0.040ms

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  var newGuy = new Knight( firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){ 
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  new Knight( firstRegimentNewbs[i], 1)
  firstRegimentKnights.push(      );
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){  -
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
    new Knight( firstRegimentNewbs[i], 1)
    firstRegimentKnights.push(
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){ █
  this.name = name;
  this.regiment = regiment;
  switch (regiment) {
    case 1:
      this.weapon = "Broadsword";
      break;
    case 2:
      this.weapon = "Claymore";
      break;
    case 3:
      this.weapon = "Longsword";
      break;
    case 5:
      this.weapon = "War Hammer";
      break;
    case 6:
      this.weapon = "Battle Axe";
      break;
    case 4:
    case 7:
    case 8:
      this.weapon = "Morning Star";
      break;
    case "King":
      this.weapon = "Excalibur";
      break;
    default:
      alert(name + " has an incorrect " +
            "regiment, Master Armourer!" +
            "\n\nNo weapon assigned!");
  }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
    firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
        case 1:
            this.weapon = "Broadsword";
            break;
        case 2:
            this.weapon = "Claymore";
            break;
        case 3:
            this.weapon = "Longsword";
            break;
        case 5:
            this.weapon = "War Hammer";
            break;
        case 6:
            this.weapon = "Battle Axe";
            break;
        case 4:
        case 7:
        case 8:
            this.weapon = "Morning Star";
            break;
        case "King":
            this.weapon = "Excalibur";
            break;
        default:
            alert(name + " has an incorrect " +
                  "regiment, Master Armourer!" +
                  "\n\nNo weapon assigned!");
    }
}
```

CONSOLE.TIME LETS US COMPARE IMPLEMENTATIONS

This timing feature can help determine the code that creates the best experience.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
    "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
    firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
        case 1:
            this.weapon = "Broadsword";
            break;
        case 2:
            this.weapon = "Claymore";
            break;
        case 3:
            this.weapon = "Longsword";
            break;
        case 5:
            this.weapon = "War Hammer";
            break;
        case 6:
            this.weapon = "Battle Axe";
            break;
        case 4:
        case 7:
        case 8:
            this.weapon = "Morning Star";
            break;
        case "King":
            this.weapon = "Excalibur";
            break;
        default:
            alert(name + " has an incorrect " +
                "regiment, Master Armourer!" +
                "\n\nNo weapon assigned!");
    }
}
```

→ Time to add 4 Knights: 0.029ms



Times will vary by browser, but ours was slightly better when not using the extra variable declaration and assignment inside the loop (a best practice!)

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
    "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
    firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];

console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```



First, we'll add a set of new guys that need to be added to the Second Regiment of Knights.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
```

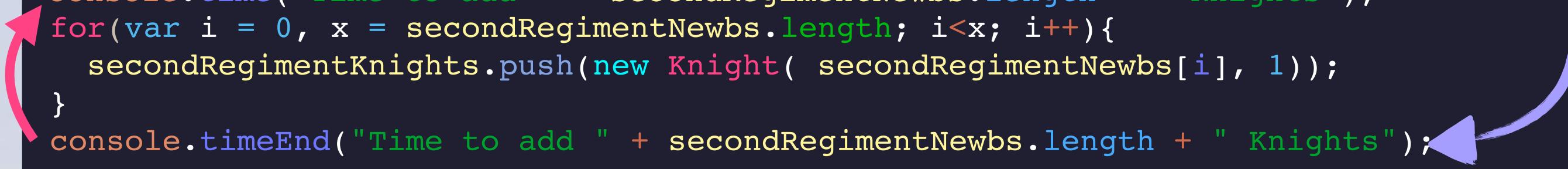
Next we'll start a timer for the total time of operation.

Both parts of our existing timer for the First Regiment additions stay in place.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
console.time("Time to add " + secondRegimentNewbs.length + " Knights");
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + secondRegimentNewbs.length + " Knights");
```



Once these new additions are done, we'll stop the second regiment timer.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");
console.time("Time to add " + secondRegimentNewbs.length + " Knights");
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + secondRegimentNewbs.length + " Knights");
console.timeEnd("Total completion time");
```

Finally, we'll wrap up the timing of the entire operation.

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Total completion time");
console.time("Time to add " + firstRegimentNewbs.length + " Knights");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + firstRegimentNewbs.length + " Knights");

console.time("Time to add " + secondRegimentNewbs.length + " Knights");
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Time to add " + secondRegimentNewbs.length + " Knights");
console.timeEnd("Total completion time");
```

→ Time to add 4 Knights: 0.031ms
→ Time to add 8 Knights: 0.063ms
→ Total completion time: 0.324ms

Remember,
setting up the
timers takes
time, too!



MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Total completion time");

for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}

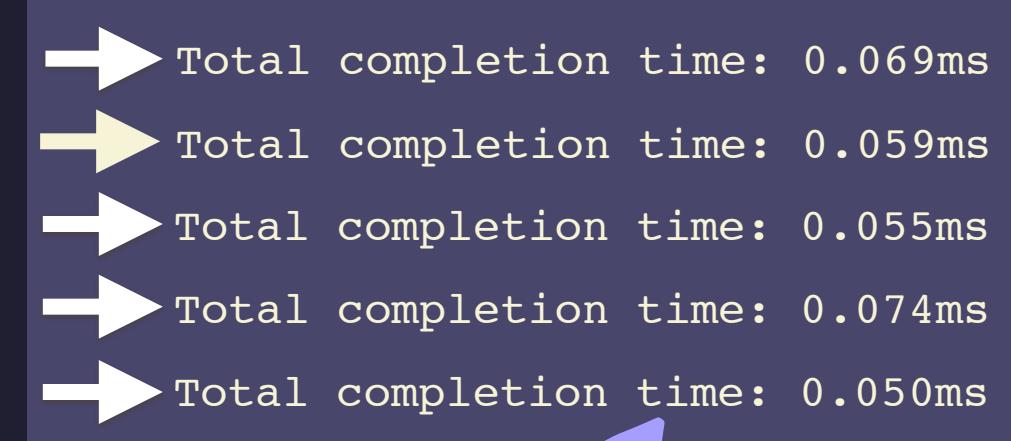
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}

console.timeEnd("Total completion time");
```

MULTIPLE TIMERS CAN RUN AT ONE TIME

Since timers are created by the label you provide, creative stacking can produce layers of time data.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn", "Bunder Ropefist",
                           "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var secondRegimentNewbs = ["Jenner Pond", "Tar Backstrand", "Cromer Treen", "Stim Lancetip",
                           "Vorn Sharpeye", "Rack Leaflets", "Bruck Valleyhome", "Arden Follower"];
var secondRegimentKnights = [ *...tons of Knight objects...* ];
console.time("Total completion time");
for(var i = 0, x = firstRegimentNewbs.length; i<x; i++){
  firstRegimentKnights.push(new Knight( firstRegimentNewbs[i], 1));
}
for(var i = 0, x = secondRegimentNewbs.length; i<x; i++){
  secondRegimentKnights.push(new Knight( secondRegimentNewbs[i], 1));
}
console.timeEnd("Total completion time");
```



→ Total completion time: 0.069ms
→ Total completion time: 0.059ms
→ Total completion time: 0.055ms
→ Total completion time: 0.074ms
→ Total completion time: 0.050ms



If we want more accuracy in our estimate, we need to use the average of multiple tests.

THE PENDANT OF PERFORMANCE

Section 5
Measuring Performance II:

Speed Averaging

JAVASCRIPT
BEST PRACTICES

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = new Date();
console.log(rightNow);
```

→ Mon Apr 10 2014 17:50:38 GMT-0500 (EST)



A new `Date` object immediately captures the current date and time, measured in milliseconds since 12:00 am, January 1st, 1970!

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = new Date();
console.log(rightNow);
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = new Date();
console.log(+rightNow);
```

...same thing as...

```
console.log(new Number(rightNow));
```

Placing a `+` unary operator in front of our `Date` object variable asks for the specific value in milliseconds.

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = new Date();
console.log(+rightNow); → console.log(new Number(rightNow));
```

→ 1392072557874



So, you know...a few.

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = new Date();
console.log(+rightNow);
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = new Date();  
+
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = +new Date();
```



Since we know we want an actual number, we can go ahead and assign the variable to be the numerical version of our new `Date` Object.

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = +new Date();
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = +new Date();  
var endTime = +new Date();
```



We can do this for a later moment
and also get a millisecond value.
You probably see what's coming...

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = +new Date();
var endTime = +new Date();
```

RETRIEVING AND USING NUMERICAL TIME DATA

To accurately generate numerical data we can manipulate, we'll first examine the JavaScript `Date` object.

```
var rightNow = +new Date();
var endTime = +new Date();
var elapsedTime = endTime - rightNow;
console.log(elapsedTime);
```

→ 87874



The difference between the two values will be the amount of time that passed between the creation of both variables.

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){  
    ↑  
    This will be the specific code we  
    want to test for performance  
    speed. We'll encapsulate it all  
    within its own function later.  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){  
    This represents whatever parameters  
    our test code needs in order to work  
    correctly. Might be an array of values,  
    or it might be a single value.  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){  
}  
  
The higher the repetitions, the  
more reliable our average speed.  
↑
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(testImplement, testParams, repetitions){  
    this.testImplement = testImplement;  
    this.testParams = testParams;  
    this.repetitions = repetitions || 10000;  
    this.average = 0;  
}
```



We make our `repetitions` parameter optional by defaulting to 10,000 executions, using our Logical Assignment best practice. We'll look at this optional nature in a bit.

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest( ) {  
    this.testImplement = testImplement;  
    this.testParams = testParams;  
    this.repetitions = repetitions || 10000;  
    this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
function SpeedTest(...){  
    this.testImplement = testImplement;  
    this.testParams = testParams;  
    this.repetitions = repetitions || 10000;  
    this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {
```



To use our best practice of adding commonly used methods to prototypes, we'll need to build the prototype itself.

```
}
```

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){
```



We'll call this inherited method to begin calculating an average speed for some test implementation.

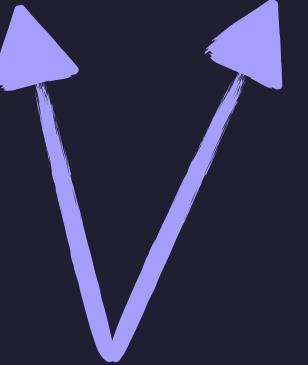
```
}
```

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
  }  
}
```



This variable will sum up all the times for all the repetitions.

These two variables will get our numerical Date objects. Notice the legibility best practice of using a comma with no extra typed var's.

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
    }  
  }  
}
```



We'll loop over the full amount
of requested repetitions.
Another best practice: no
repetitive property access!

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
    }  
  }  
}
```



Start the clock for this individual repetition!

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
      this.testImplement( this.testParams );  
    }  
  }  
}
```



Here's where we run the
code for this repetition!

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
      this.testImplement( this.testParams );  
      endTime = +new Date();  
    }  
  }  
}
```



Stop the clock!

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
      this.testImplement( this.testParams );  
      endTime = +new Date();  
      sumTimes += endTime - beginTime;  
    }  
  }  
}
```



Here we add in the individual time spent on this particular test.

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
      this.testImplement( this.testParams );  
      endTime = +new Date();  
      sumTimes += endTime - beginTime;  
    }  
    this.average = sumTimes / this.repetitions;  
  }  
}
```



Average is sum of all times,
divided by repetitions.

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

ONWARD! NOW TO CREATE A SPEED TEST CLASS

We'll use our Date object approach to calculate a more accurate performance over many code repetitions.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
      this.testImplement( this.testParams );  
      endTime = +new Date();  
      sumTimes += endTime - beginTime;  
    }  
    this.average = sumTimes / this.repetitions;  
    return console.log("Average execution across " +  
                      this.repetitions + ":" +  
                      this.average);  
  }  
}
```

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

We return a message
with the average time.

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
SpeedTest.prototype = {  
  startTest: function(){  
    var beginTime, endTime, sumTimes = 0;  
    for (var i = 0, x = this.repetitions; i < x; i++){  
      beginTime = +new Date();  
      this.testImplement( this.testParams );  
      endTime = +new Date();  
      sumTimes += endTime - beginTime;  
    }  
    this.average = sumTimes / this.repetitions;  
    return console.log("Average execution across " +  
                      this.repetitions + ":" +  
                      this.average);  
  }  
}
```

```
function SpeedTest(...){  
  this.testImplement = testImplement;  
  this.testParams = testParams;  
  this.repetitions = repetitions || 10000;  
  this.average = 0;  
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *....tons of Knight objects...* ];

for(var i = 0; i < firstRegimentNewbs.length; i++){
  var newGuy = new Knight(firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}
```



To test this block, we need to be able to pass all of it into the SpeedTest constructor as its own function.



```
var noBPTest = new SpeedTest( );
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                       this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *....tons of Knight objects...* ];

for(var i = 0; i < firstRegimentNewbs.length; i++){
  var newGuy = new Knight(firstRegimentNewbs[i], 1);
  firstRegimentKnights.push(newGuy);
}

var noBPtest = new SpeedTest( );
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var noBP = function () {
  for(var i = ; i < firstRegimentNewbs.length; i++){
    var newGuy = new Knight(firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
  }
};
```

If we wrap the code in its own function expression and assign it to a variable, we can easily pass it around as a "set" of code to be tested.

```
var noBPtest = new SpeedTest(noBP, );
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                       this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *....tons of Knight objects...* ];

var noBP = function ( listOfParams ) {
  for(var i = 0; i < firstRegimentNewbs.length; i++){
    var newGuy = new Knight(firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
  }
};
```

Since our `SpeedTest` lumps all the test code's important parameters into one `testParams` property, we'll need to modify our function expression and important data to use just one ARRAY of important stuff.

```
var noBPtest = new SpeedTest(noBP, );
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];

var noBP = function ( listOfParams ) {
  for(var i = 0; i < firstRegimentNewbs.length; i++){
    var newGuy = new Knight(firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
  }
};

var noBPtest = new SpeedTest(noBP, );
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < firstRegimentNewbs.length; i++ ){
    var newGuy = new Knight(firstRegimentNewbs[i], 1);
    firstRegimentKnights.push(newGuy);
  }
};
```

Now that we have an array of the parameters that `noBP` will need, we have to replace the existing names in the function with references to our single parameter array.

```
var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight( listOfParams[0][i], 1 );
    .push(newGuy);
  }
};
```

Since `firstRegimentNewbs` is in the zeroth index of our new parameter array, we'll use that index to access its contents.

```
var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];

var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight( listOfParams[0][i], 1 );
    .push(newGuy);
  }
};

var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```



Similarly, we'll use the next index for `firstRegimentKnights`.

```
var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];

var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};

var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

We'll first build some code that lacks some Best Practices and test its speed.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];

var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};

var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];

var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};

var noBPtest = new SpeedTest(noBP, listsForTests);
```



In creating our new `SpeedTest` object, we're first opting to NOT pass in a `repetitions` amount, and thus receive our logical assignment default of 10,000.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams,
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];

var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};

var noBPtest = new SpeedTest(noBP, listsForTests);
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests);
noBPtest.startTest();
```



→ Average execution across 10000: 0.0041

Notice that we have about 10 times
a speed improvement over the
estimations from `console.time`.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests);
noBPtest.startTest();
```

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```

```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```



LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now we're ready to test the speed of our prepared code sample!

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



→ Average execution across 100000: 0.00478

Our newer estimate brings us ever closer to the truth.

```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}

SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i < listOfParams[0].length; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
                      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0, x = listOfParams[0].length; i < x; i++ ){
    var newGuy = new Knight(listOfParams[0][i], 1);
    listOfParams[1].push(newGuy);
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for( var i = 0, x = listOfParams[0].length; i < x; i++ ){
    new Knight(listOfParams[0][i], 1)
    listOfParams[1].push( );
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for ( var i = 0, x = this.repetitions; i < x; i++ ){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for(var i = 0, x = listOfParams[0].length; i < x; i++){
    new Knight(listOfParams[0][i], 1)
    listOfParams[1].push(
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                          "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var noBP = function ( listOfParams ) {
  for(var i = 0, x = listOfParams[0].length; i < x; i++){
    listOfParams[1].push(new Knight(listOfParams[0][i], 1));
  }
};
```

```
var noBPtest = new SpeedTest(noBP, listsForTests, 100000);
noBPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
                           "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var BP = function ( listOfParams ) {
  for(var i = 0, x = listOfParams[0].length; i < x; i++){
    listOfParams[1].push(new Knight(listOfParams[0][i], 1));
  }
};
```

```
var BPtest = new SpeedTest(noBP, listsForTests, 100000);
BPtest.startTest();
```



```
function SpeedTest(...){
  this.testImplement = testImplement;
  this.testParams = testParams;
  this.repetitions = repetitions || 10000;
  this.average = 0;
}
```



```
SpeedTest.prototype = {
  startTest: function(){
    var beginTime, endTime, sumTimes = 0;
    for (var i = 0, x = this.repetitions; i < x; i++){
      beginTime = +new Date();
      this.testImplement( this.testParams );
      endTime = +new Date();
      sumTimes += endTime - beginTime;
    }
    this.average = sumTimes / this.repetitions;
    return console.log("Average execution across " +
      this.repetitions + ":" + this.average);
  }
}
```

LET'S TEST AN IMPLEMENTATION WITH OUR CLASS

Now let's add some Best Practices to see if we can shorten the time.

```
var firstRegimentNewbs = ["Grimble Horsehead", "Jark Winterborn",
    "Bunder Ropefist", "Ernst Breadbaker"];
var firstRegimentKnights = [ *...tons of Knight objects...* ];
var listsForTests = [ firstRegimentNewbs, firstRegimentKnights ];
```

```
var BP = function ( listOfParams ){
    for(var i = 0, x = listOfParams[0].length; i < x; i++){
        listOfParams[1].push(new Knight(listOfParams[0][i], 1));
    }
};
```

```
var BPtest = new SpeedTest(noBP, listsForTests, 100000);
BPtest.startTest();
```



→ Average execution across 100000: 0.00274

Significantly better execution, especially
if this process were replicated a few
thousand times in a program!



```
function SpeedTest(...){
    this.testImplement = testImplement;
    this.testParams = testParams;
    this.repetitions = repetitions || 10000;
    this.average = 0;
}

SpeedTest.prototype = {
    startTest: function(){
        var beginTime, endTime, sumTimes = 0;
        for (var i = 0, x = this.repetitions; i < x; i++){
            beginTime = +new Date();
            this.testImplement( this.testParams );
            endTime = +new Date();
            sumTimes += endTime - beginTime;
        }
        this.average = sumTimes / this.repetitions;
        return console.log("Average execution across " +
            this.repetitions + ":" + this.average);
    }
}
```

THE CRYSTAL OF CAUTION

Section 1
Careful Comparisons

JAVASCRIPT
BEST PRACTICES

NOT ALL EQUALITY COMPARISONS ARE EQUAL

The triple-equals comparator compares both type AND contents.

```
'4' == 4
```



→ true

Double>equals tries to help with
type coercion ... but it's not always
the help we want.

```
'4' === 4
```



→ false

Triple>equals makes sure data
have both similar type and
contents.

NOT ALL EQUALITY COMPARISONS ARE EQUAL

The triple-equals comparator compares both type AND contents.

```
'4' == 4
```

→ true

```
true == 1
```

→ true

```
false == 0
```



→ true

```
'4' === 4
```

→ false

```
true === 1
```

→ false

```
false === 0
```



→ false

By historical convention within the field of computing, `true` and `false` traditionally evaluate to 1 and 0, respectively.

Triple equals don't care! It first sees a Boolean and a number and says: yo, these ain't the same.

NOT ALL EQUALITY COMPARISONS ARE EQUAL

The triple-equals comparator compares both type AND contents.

```
' 4 ' == 4
```

→ true

```
' 4 ' === 4
```

→ false

```
true == 1
```

→ true

```
true === 1
```

→ false

```
false == 0
```

→ true

```
false === 0
```

→ false

```
"\n \n \t" == 0
```

→ true

```
"\n \n \t" === 0
```

→ false



Uhh, what?

TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `====` ensures comparison reliability in environments where data may have multiple or unknown types.

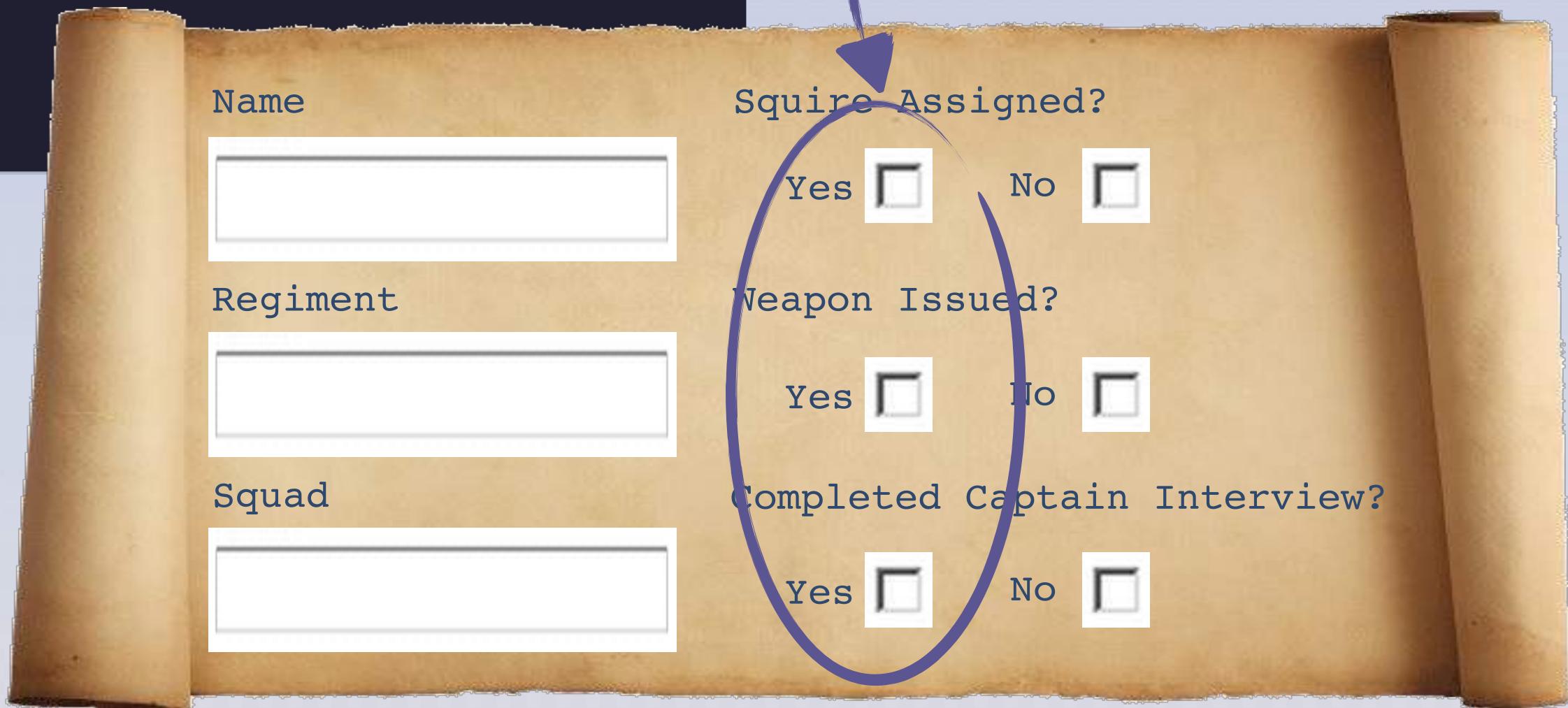
```
function countEntries( KnightResponses, value ){
  var count = 0, x = KnightResponses.length;
  for(var i = 0; i<x; i++) {
    if(KnightResponses[i] === value){
      count++;
    }
  }
  return count;
}
```

Name	Squire Assigned?
<input type="text"/>	Yes <input type="checkbox"/> No <input type="checkbox"/>
Regiment	Weapon Issued?
<input type="text"/>	Yes <input type="checkbox"/> No <input type="checkbox"/>
Squad	Completed Captain Interview?
<input type="text"/>	Yes <input type="checkbox"/> No <input type="checkbox"/>

TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `====` ensures comparison reliability in environments where data may have multiple or unknown types.

```
function countEntries( KnightResponses, value ){  
  var count = 0, x = KnightResponses.length;  
  for(var i = 0; i<x; i++) {  
    if(KnightResponses[i] === value){  
      count++;  
    }  
  }  
  return count;  
}
```



TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `==` ensures comparison reliability in environments where data may have multiple or unknown types.

```
function countEntries( KnightResponses, value ){
  var count = 0, x = KnightResponses.length;
  for(var i = 0; i<x; i++) {
    if(KnightResponses[i] == value){
      count++;
    }
  }
  return count;
}
```

Name	Squire Assigned?	
<input type="text" value="Jason Millhouse"/>	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Regiment	Weapon Issued?	
<input type="text" value="1"/>	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Squad	Completed Captain Interview?	
<input type="text" value="12"/>	Yes <input type="checkbox"/>	No <input checked="" type="checkbox"/>

TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `==` ensures comparison reliability in environments where data may have multiple or unknown types.

```
function countEntries( KnightResponses, value ){
  var count = 0, x = KnightResponses.length;
  for(var i = 0; i<x; i++) {
    if(KnightResponses[i] == value){
      count++;
    }
  }
  return count;
}
```

The image shows a parchment scroll with a form. The fields are:

Name	Jason Millhouse	Squire Assigned?	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Regiment	1	Weapon Issued?	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Squad	12	Completed Captain Interview?	Yes <input type="checkbox"/>	No <input checked="" type="checkbox"/>

```
var fields = ["Jason Millhouse", "1", "12", true, true, false];
```

TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `==` ensures comparison reliability in environments where data may have multiple or unknown types.

```
function countEntries( KnightResponses, value ){  
  var count = 0, x = KnightResponses.length;  
  for(var i = 0; i<x; i++) {  
    if(KnightResponses[i] == value){  
      count++;  
    }  
  }  
  return count;  
}
```

The form contains the following data:

Name	Squire Assigned?
Jason Millhouse	Yes <input checked="" type="checkbox"/> No <input type="checkbox"/>
Regiment	Weapon Issued?
1	Yes <input checked="" type="checkbox"/> No <input type="checkbox"/>
Squad	Completed Captain Interview?
12	Yes <input type="checkbox"/> No <input checked="" type="checkbox"/>

```
var fields = ["Jason Millhouse", "1", "12", true, true, false];  
var numCompletedTasks = countEntries( fields, true );  
console.log( numCompletedTasks );
```

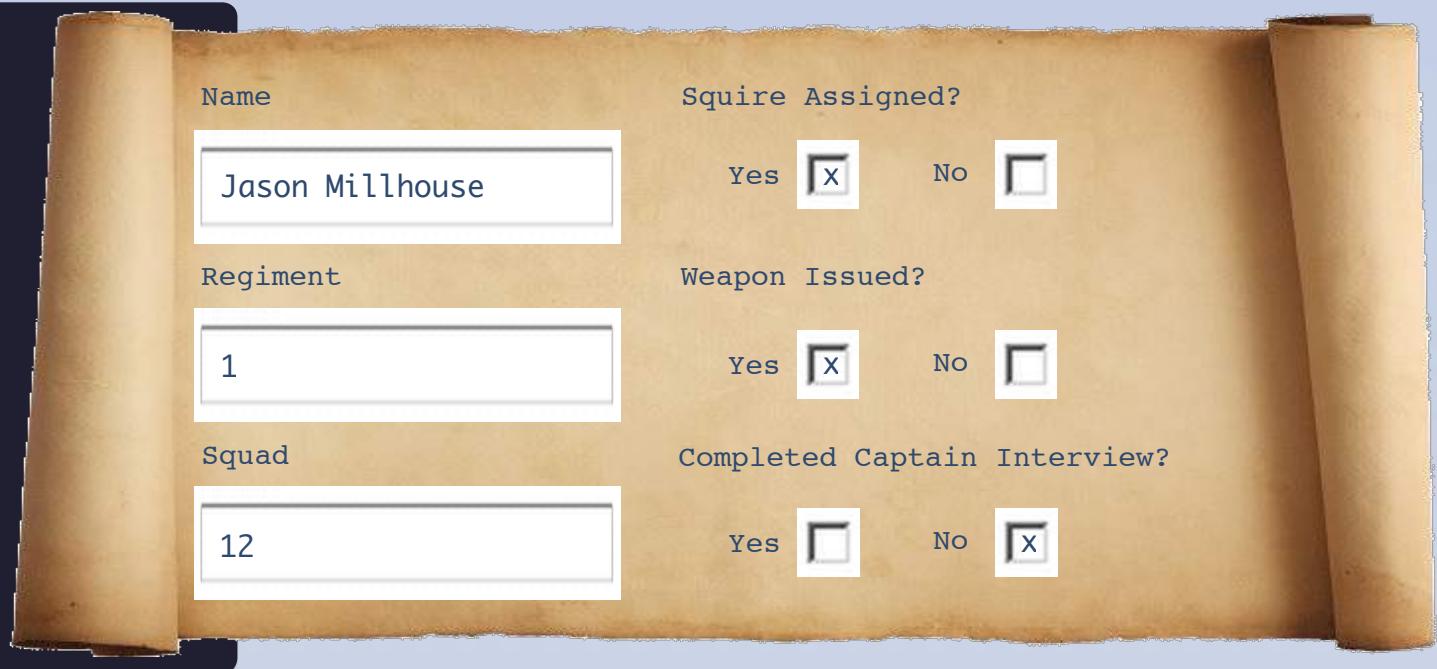
→ 3



TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `==` ensures comparison reliability in environments where data may have multiple or unknown types.

```
function countEntries( KnightResponses, value ){  
  var count = 0, x = KnightResponses.length;  
  for(var i = 0; i<x; i++) {  
    if(KnightResponses[i] == value){  
      count++;  
    }  
  }  
  return count;  
}
```



```
var fields = ["Jason Millhouse", "1", "12", true, true, false];  
var numCompletedTasks = countEntries( fields, true );  
console.log( numCompletedTasks );
```

"1" == true
→ true

✗ → 3

TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `====` ensures comparison reliability in environments where data may have multiple or unknown types.

```
function countEntries( KnightResponses, value ){
  var count = 0, x = KnightResponses.length;
  for(var i = 0; i<x; i++) {
    if(KnightResponses[i] === value){
      count++;
    }
  }
  return count;
}
```

Name	Jason Millhouse	Squire Assigned?	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Regiment	1	Weapon Issued?	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Squad	12	Completed Captain Interview?	Yes <input type="checkbox"/>	No <input checked="" type="checkbox"/>

```
var fields = ["Jason Millhouse", "1", "12", true, true, false];
var numCompletedTasks = countEntries( fields, true );
console.log( numCompletedTasks );
```

TRIPLE-EQUALS SEEKS A “STRICT” EQUALITY

A `====` ensures comparison reliability in environments where data may have multiple or unknown types.

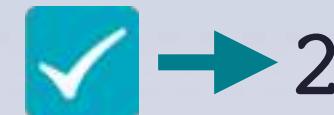
```
function countEntries( KnightResponses, value ){  
  var count = 0, x = KnightResponses.length;  
  for(var i = 0; i<x; i++) {  
    if(KnightResponses[i] === value){  
      count++;  
    }  
  }  
  return count;  
}
```

The form contains the following data:

Name	Jason Millhouse	Squire Assigned?	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Regiment	1	Weapon Issued?	Yes <input checked="" type="checkbox"/>	No <input type="checkbox"/>
Squad	12	Completed Captain Interview?	Yes <input type="checkbox"/>	No <input checked="" type="checkbox"/>

```
var fields = ["Jason Millhouse", "1", "12", true, true, false];  
var numCompletedTasks = countEntries( fields, true );  
console.log( numCompletedTasks );
```

"1" === true
→ false



WHAT IF WE NEED TO VERIFY AN OBJECT'S "CLASS"?

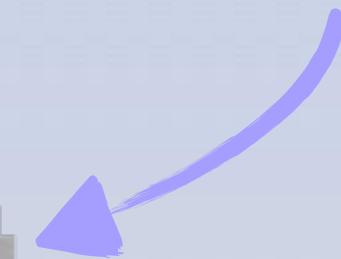
Sometimes we must make sure an Object has been built by a specific Constructor, or has a specific Prototype.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



But Knights only get Chain Mail, duh. How can we find which armor objects are Chain Mail?

```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

WHAT IF WE NEED TO VERIFY AN OBJECT'S "CLASS"?

Sometimes we must make sure an Object has been built by a specific Constructor, or has a specific Prototype.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```

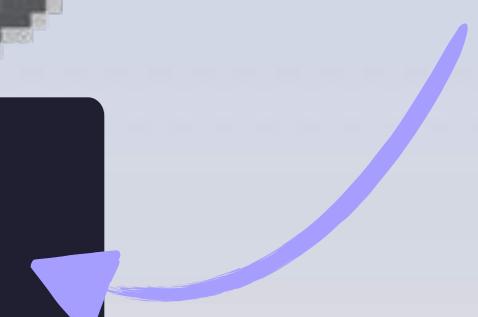


```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

If we just loop over the armor list, passing out armor to Knights, some might end up with **LeatherArmor** objects ... or worse!



THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ) {  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ) {  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ) {
```



We'll pass in both lists.

```
}
```

THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ) {  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ) {  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ) {  
    var x = knights.length;  
    var y = armorAvail.length;
```



Cache the array
lengths for efficiency.

THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ) {  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ) {  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ) {  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++) {
```



We'll loop over every **Knight** that needs some armor.

```
}
```

THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            }
```



For each `Knight`, we'll then check through the armor array to find some Chain Mail.

THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                }  
            }  
        }  
    }
```



Here's where we use `instanceof` to see if the currently examined armor is a `ChainMail` object.

THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
            }  
        }  
    }  
}
```

As soon as we find a `ChainMail`,
we splice it out of the `armor`
array and give it to the `Knight`.



THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
            }  
        }  
    }  
}
```

Since we modified the armor array, we adjust our cached length.

THE INSTANCEOF OPERATOR HELPS IDENTIFY OBJECTS

Use this operator to ensure you're examining precisely the type of object that your code expects.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

Since the current `Knight` has some armor now, we can break out of the inner-most for-loop.



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ *...the Armory's mixed up list  
                  of all available armor Objects...* ];  
var newbs = [ *...a list of Knight Objects...* ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [  
];  
var newbs = [  
];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  ChainMail{...},  
                  LeatherArmor{...}  
                  ChainMail{...}  
                ];  
  
var newbs = [ Knight{...}, Knight{...} ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );
```

TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  ChainMail{...},  
                  LeatherArmor{...}  
                  ChainMail{...}  
                ];  
var newbs = [ Knight{...}, Knight{...} ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );
```



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  LeatherArmor{...}  
                  ChainMail{...}  
                ];  
var newbs = [ Knight{...}, Knight{...} ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );
```



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  LeatherArmor{...}  
                  ChainMail{...}  
                ];
```

```
var newbs = [ Knight{...}, Knight{...} ];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );
```



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```

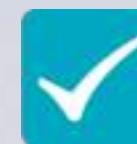


```
var armorList = [ LeatherArmor{...},  
                  LeatherArmor{...}  
                  ChainMail{...}  
                ];
```

```
var newbs = [ Knight{...}, Knight{...} ]
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );
```



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  LeatherArmor{...}  
                ];
```

```
var newbs = [ Knight{...}, Knight{...} ]
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );
```



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  LeatherArmor{...}  
                ];  
var newbs = [ Knight{...}, Knight{...}];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );  
console.log( armorList );
```



[LeatherArmor{...}, LeatherArmor{...}]

No more `ChainMail` objects!



TIME TO GIVE SOME NEWBS SOME CHAIN MAIL

Let's fill the lists with a short example and check out the results of our armor identification process.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



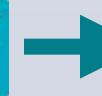
```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
var armorList = [ LeatherArmor{...},  
                  LeatherArmor{...}  
                ];  
var newbs = [ Knight{...}, Knight{...}];
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
assignKnightsArmor( newbs, armorList );  
console.log( newbs[0] );
```



{ *bunch of properties*, armor: ChainMail{...} }

Now our Knights have their armor.

ALSO USE `INSTANCEOF` TO CHECK THE ENTIRE INHERITANCE CHAIN

An object is an “instance” of all of the prototypes from which it inherits properties.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}
```



```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

ALSO USE INSTANCEOF TO CHECK THE ENTIRE INHERITANCE CHAIN

An object is an “instance” of all of the prototypes from which it inherits properties.

```
function LeatherArmor ( bodyStyle, numBuckles,  
                        numSpaulders ){  
    this.bodyStyle = bodyStyle;  
    this.numBuckles = numBuckles;  
    this.numSpaulders = numSpaulders;  
}  
LeatherArmor.prototype = Object.create(Armor.prototype);
```



```
function ChainMail ( metal, linkDiameter,  
                     hasHood, skirtLength ){  
    this.metal = metal;  
    this.linkDiameter = linkDiameter;  
    this.hasHood = hasHood;  
    this.skirtLength = skirtLength;  
}  
ChainMail.prototype = Object.create(Armor.prototype);
```



```
function Armor ( location ){  
    this.location = location;  
}  
Armor.prototype = {  
    putOn: function(){  
        alert("Your armor is on.");  
    }  
}
```

```
function assignKnightsArmor ( knights, armorAvail ){  
    var x = knights.length;  
    var y = armorAvail.length;  
    for(var i = 0; i < x; i++){  
        for(var j = 0; j < y; j++){  
            if( armorAvail[j] instanceof ChainMail){  
                knights[i].armor = armorAvail.splice(j, 1)[0];  
                y--;  
                break;  
            }  
        }  
    }  
}
```

```
var kingsMail = new ChainMail("gold", 2, true, 36);  
console.log( kingsMail instanceof Armor );
```

Useful if you need to make sure an
Object actually has access to
properties BEFORE trying to call them!

→ true

THE CRYSTAL OF CAUTION

Section 2
Exception Handling

JAVASCRIPT
BEST PRACTICES

EXCEPTIONS ARE RUN-TIME ERRORS

Let's first examine the difference between a syntax error and a run-time error.

ALARM.JS

```
var alarm = "Dragons approach!";
alert(alarm)
```



A syntax error will not make it past the interpreter. It will be marked as invalid Javascript and reported in the JS console.

ALARM2.JS

```
alert(alarm);
```



This code is acceptable JavaScript, but it runs and finds that `alarm` has had no declaration by the time the `alert` is reached. This “run-time error” is the sort we want to catch and control.

EXCEPTIONS ARE RUN-TIME ERRORS

Let's first examine the difference between a syntax error and a run-time error.

ALARM2.JS

```
alert(alarm);
```

JAVASCRIPT
BEST PRACTICES

EXCEPTIONS ARE RUN-TIME ERRORS

Let's first examine the difference between a syntax error and a run-time error.

```
alert(alarm);
```

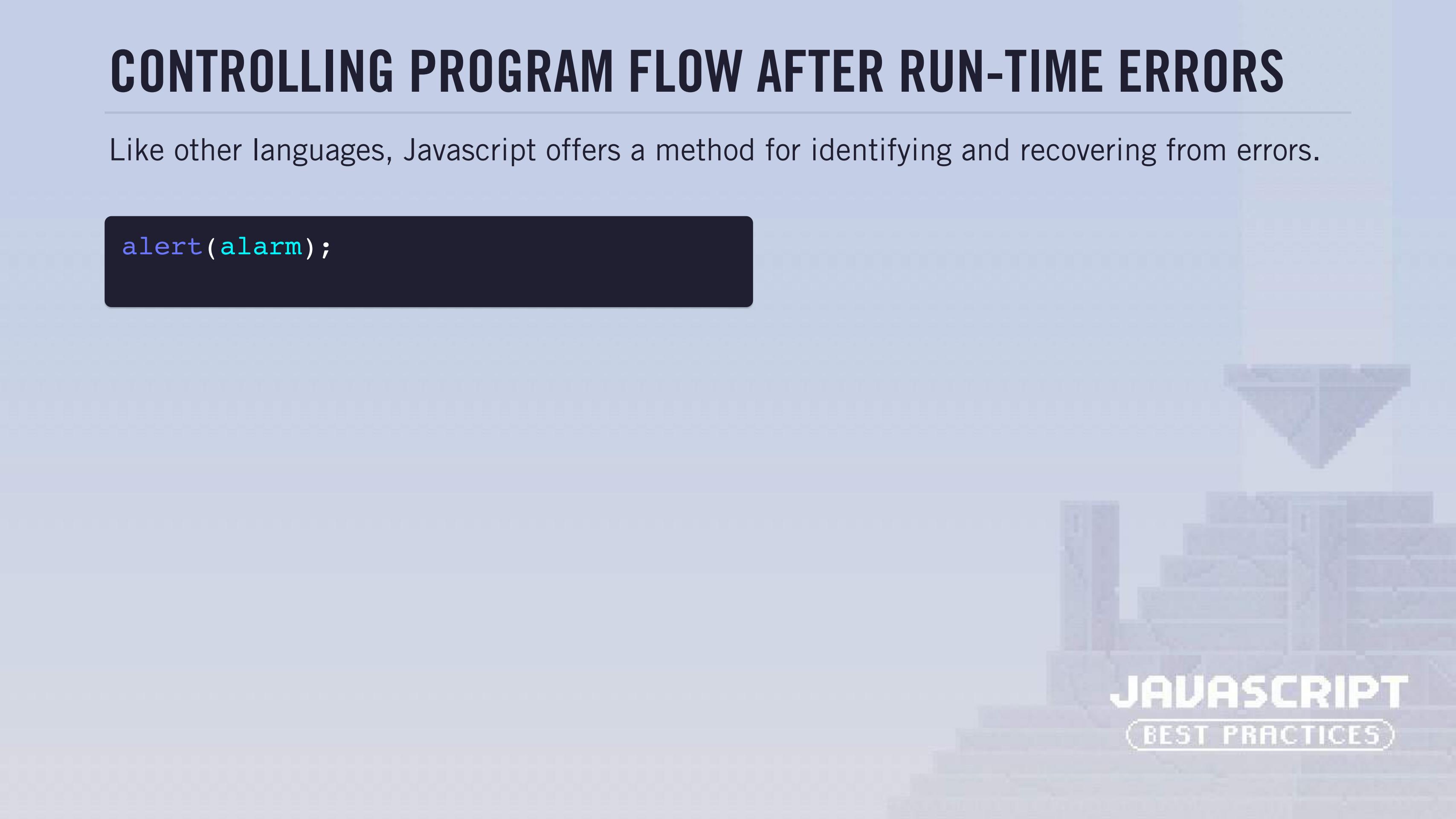


JAVASCRIPT
BEST PRACTICES

CONTROLLING PROGRAM FLOW AFTER RUN-TIME ERRORS

Like other languages, Javascript offers a method for identifying and recovering from errors.

```
alert(alarm);
```



JAVASCRIPT
BEST PRACTICES

CONTROLLING PROGRAM FLOW AFTER RUN-TIME ERRORS

Like other languages, Javascript offers a method for identifying and recovering from errors.

```
try {  
  alert(alarm);  
}
```

A `try` block is like a test zone for your code. Here, if all goes smoothly and a global `alarm` variable happens to exist when it runs, we're good to go!

CONTROLLING PROGRAM FLOW AFTER RUN-TIME ERRORS

Like other languages, Javascript offers a method for identifying and recovering from errors.

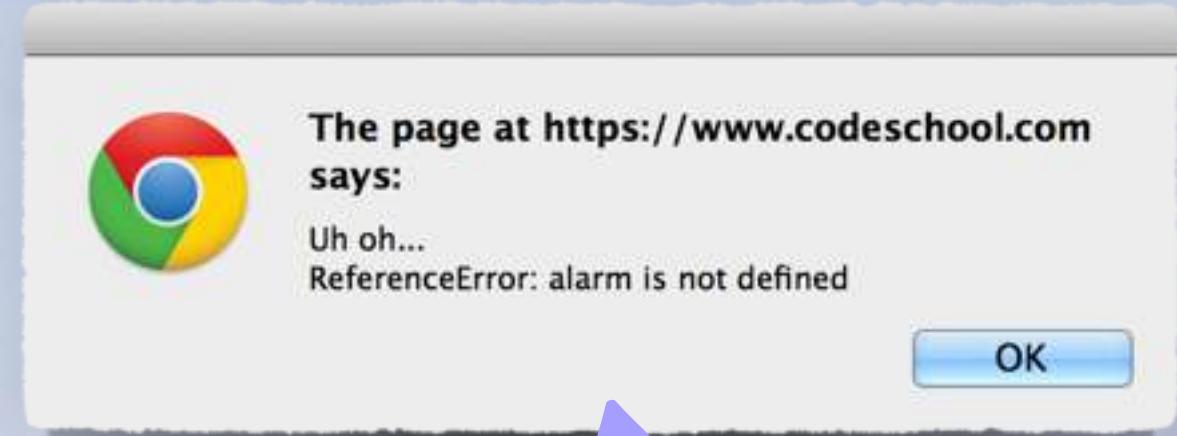
```
try {  
    alert(alarm);  
} catch (error) {  
    alert("Uh oh...\n" + error);  
}
```

If something goes wrong, however, the `try` block “throws” an error message containing details over to its buddy, the `catch` block.

CONTROLLING PROGRAM FLOW AFTER RUN-TIME ERRORS

Like other languages, Javascript offers a method for identifying and recovering from errors.

```
try {  
    alert(alarm);  
} catch (error) {  
    alert("Uh oh...\n" + error);  
}
```



As a parameter object within the catch block, this error can be used in messaging, or even in conditions to take specific action.

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

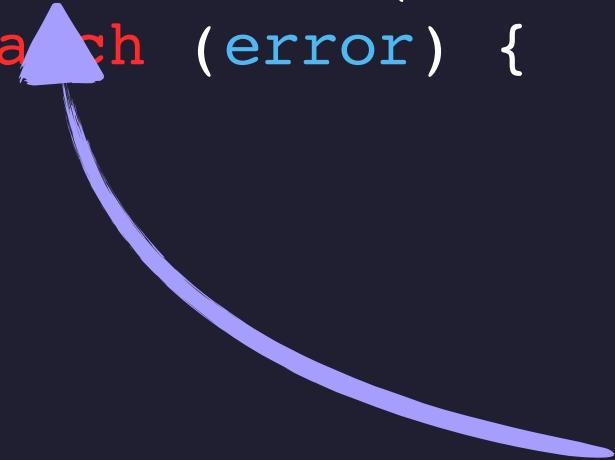
JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {  
}  
} catch (error) {  
}  
}
```

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {  
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];  
  list.concat(newHallOfFame);  
} catch (error) {  
}  
}
```



Let's say we wanted to add some new Knights to the Hall of Fame, whose existing members are held in a global array called `list`... we hope. What errors types might we expect?

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {  
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];  
  list.concat(newHallOfFame);  
} catch (error) {
```



If, for some reason, `list` has not been defined yet, or the Master Hall of Fame has been redefined, we'll get a `ReferenceError`.

```
}
```

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {  
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];  
  list.concat(newHallOfFame);  
} catch (error) {  
}  
}
```



If `list` is not an array or a string, there won't be a `concat` method to call, and we'll get a `TypeError`.

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

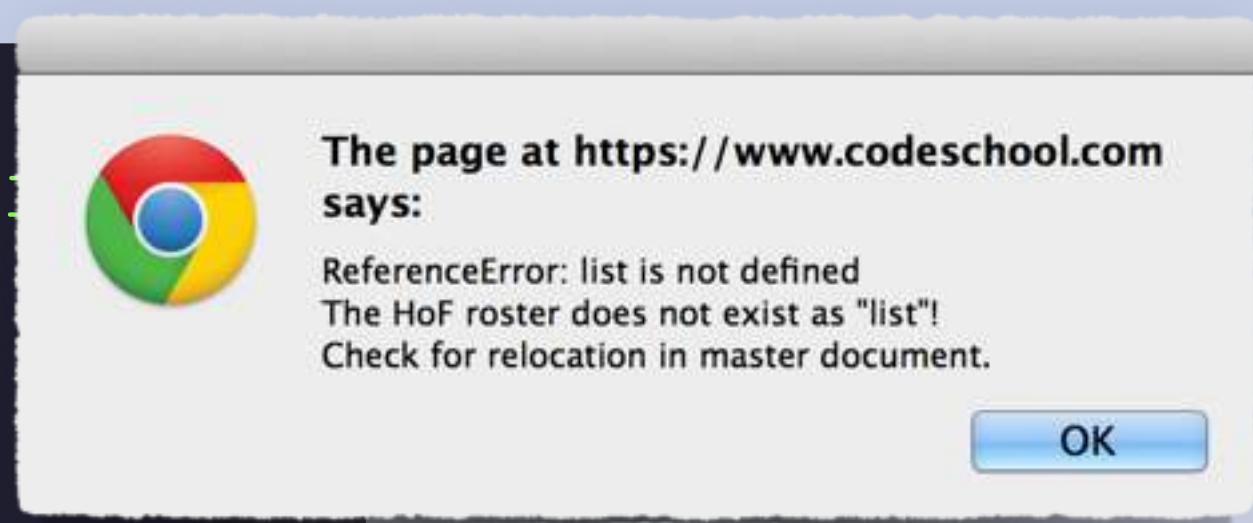
JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"! \n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Talon"];
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

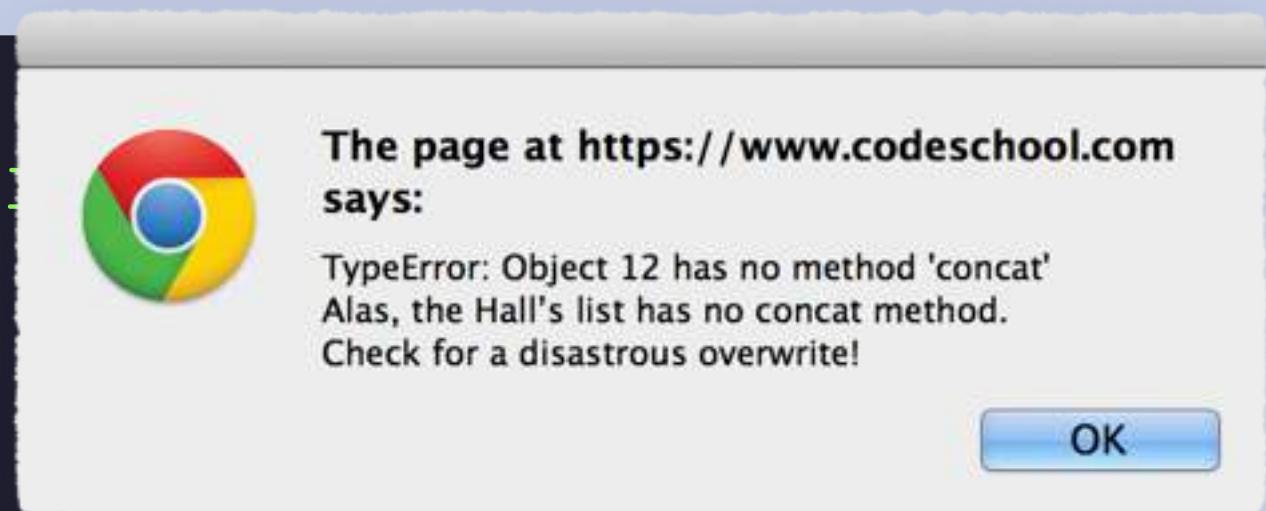


```
var hall = [ ... ];
```

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Talon"];
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```



```
list = 12;
```

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  list.concat(newHallOfFame); ←
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

What if `list` turns out to be a string? This code would run just fine and avoid the `catch` block, but likely not as expected.

"Jar Treen\nMaximo Rarter\nPol Grist" + ["Dask Frostring", "Render Tallborn"]

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  list.concat(newHallOfFame); ←
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

What if `list` turns out to be a string? This code would run just fine and avoid the `catch` block, but likely not as expected.

ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  list.concat(newHallOfFame); ←
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

What if `list` turns out to be a string? This code would run just fine and avoid the `catch` block, but likely not as expected.

"Jar Treen\nMaximo Rarter\nPol GristDask Frostring, Render Tallborn"



ERROR TYPES ALLOW US TO TAKE MORE INFORMED ACTION

JavaScript can throw multiple useful error types that allow a catch block to “handle” errors more gracefully.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

JS DOESN'T PINPOINT EVERY “ERROR” THAT MIGHT OCCUR

Use conditionals and the `throw` keyword to craft the right exception scenario based on your expectations.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  if (list === undefined){
    throw new ReferenceError();
  }
  if (!(list instanceof Array) === false){
    throw new TypeError();
  }
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, the Hall's list has no concat method.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

As soon as any `throw` is reached, `try` passes control to `catch`.

We should modify this message to be more accurate.

JS DOESN'T PINPOINT EVERY “ERROR” THAT MIGHT OCCUR

Use conditionals and the `throw` keyword to craft the right exception scenario based on your expectations.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  if (list === undefined){
    throw new ReferenceError();
  }
  if (!(list instanceof Array) === false){ ←
    throw new TypeError();
  }
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, \"list\" exists, but is no longer an Array.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

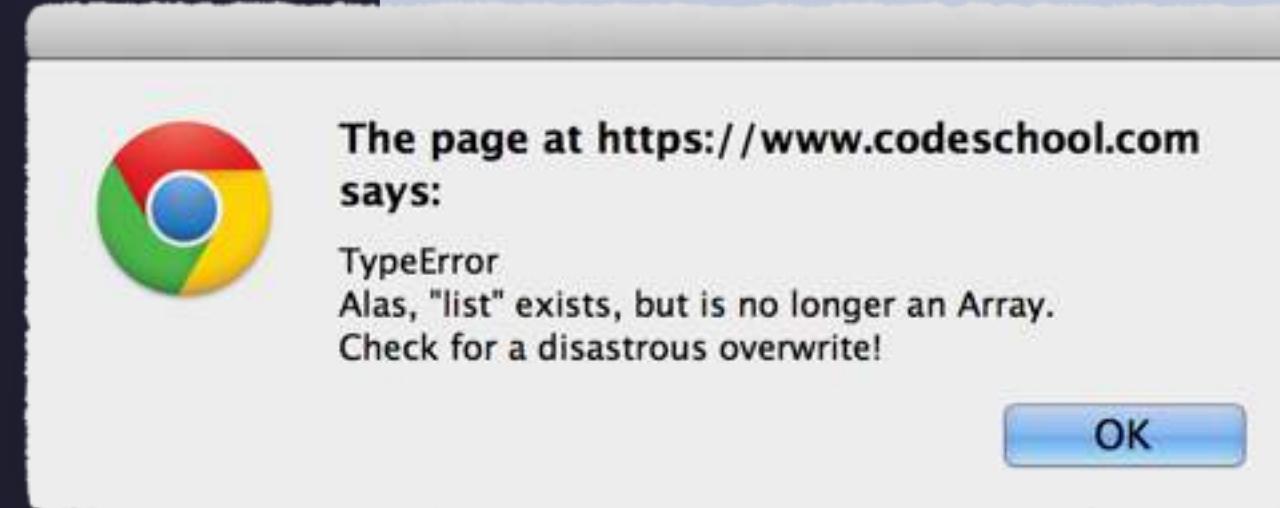
With more clearly defined problems, your program can provide more accurate debugging instructions for all cases that you might expect.

JS DOESN'T PINPOINT EVERY “ERROR” THAT MIGHT OCCUR

Use conditionals and the `throw` keyword to craft the right exception scenario based on your expectations.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  if (list === undefined){
    throw new ReferenceError();
  }
  if (!(list instanceof Array) === false){
    throw new TypeError();
  }
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, \"list\" exists, but is no longer an Array.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

list = "Jar Treen ..."; 



```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  if (list === undefined){
    throw new ReferenceError();
  }
  if (!(list instanceof Array) === false){
    throw new TypeError();
  }
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, \"list\" exists, but is no longer an Array.\n" +
      "Check for a disastrous overwrite!");
  }
}
```

WE MAY WANT TO TAKE SOME ACTION REGARDLESS OF ERROR

The `finally` block follows try/catch, and holds code that should execute whether any errors were present or not.

```
try {
  var newHallOfFame = ["Dask Frostring", "Render Tallborn"];
  if (list === undefined){
    throw new ReferenceError();
  }
  if (!(list instanceof Array) === false){
    throw new TypeError();
  }
  list.concat(newHallOfFame);
} catch (error) {
  if (error instanceof ReferenceError){
    alert(error + "\n" +
      "The HoF roster does not exist as \"list\"!\\n" +
      "Check for relocation in master document.");
  }
  if (error instanceof TypeError){
    alert(error + "\n" +
      "Alas, \"list\" exists, but is no longer an Array.\\n" +
      "Check for a disastrous overwrite!");
  }
}
```

```
  finally {
    console.log(list);
  }
```



The `finally` block will execute whether `try` met with success or failure. Now we'll know exactly what `list` contained at the end of the operation, if anything.

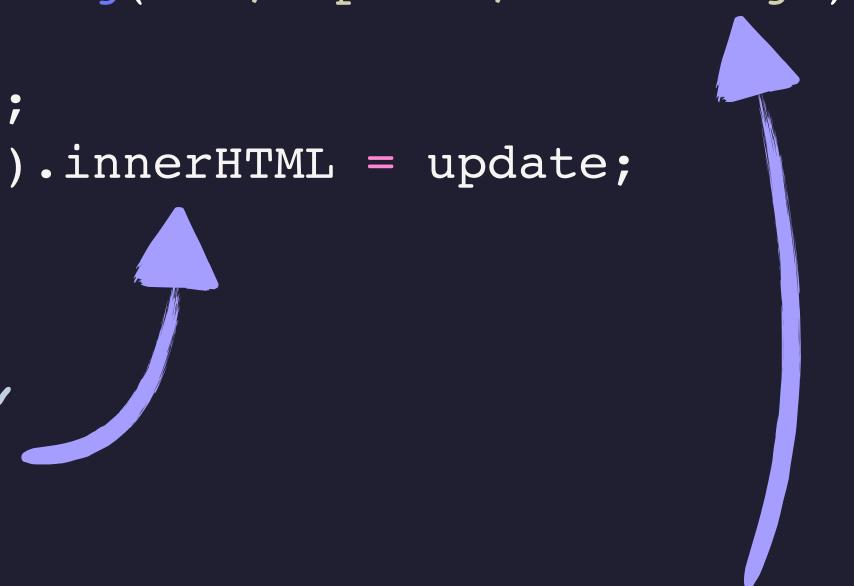
WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){  
  try{  
    var newElement = undefined;  
    document.getElementById(id).innerHTML = update;  
  }  
}
```

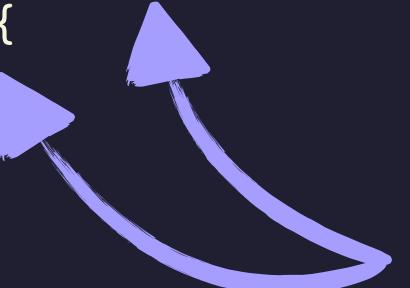
If the element is found,
we'll just update it's
inner HTML.

We'll make the existing
inner HTML optional as an
extra search possibility if
the `id` isn't found.



WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){  
  try{  
    var newElement = undefined;  
    document.getElementById(id).innerHTML = update;  
  } catch (error){  
    try{  
  
      Can't find it? Let's try  
      something else first. We may  
      know its existing inner HTML  
      instead of the node's ID.  
    }  
  }  
}
```

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i < x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
      }
    }
  }
}
```

...and then search each for a matching inner HTML. If we find a match, we'll change it and exit.

We'll get all the document elements (ugh)...

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i < x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
      }
    }
  }
}
```

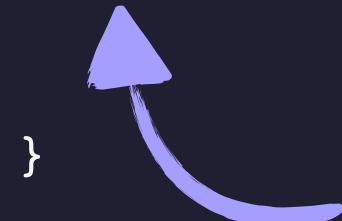


Since we don't have the right id, we'll also record it in the `id` variable, if it exists ... `undefined` otherwise.

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i < x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
        if(i === x){
          }
        }
      }
    }
}
```

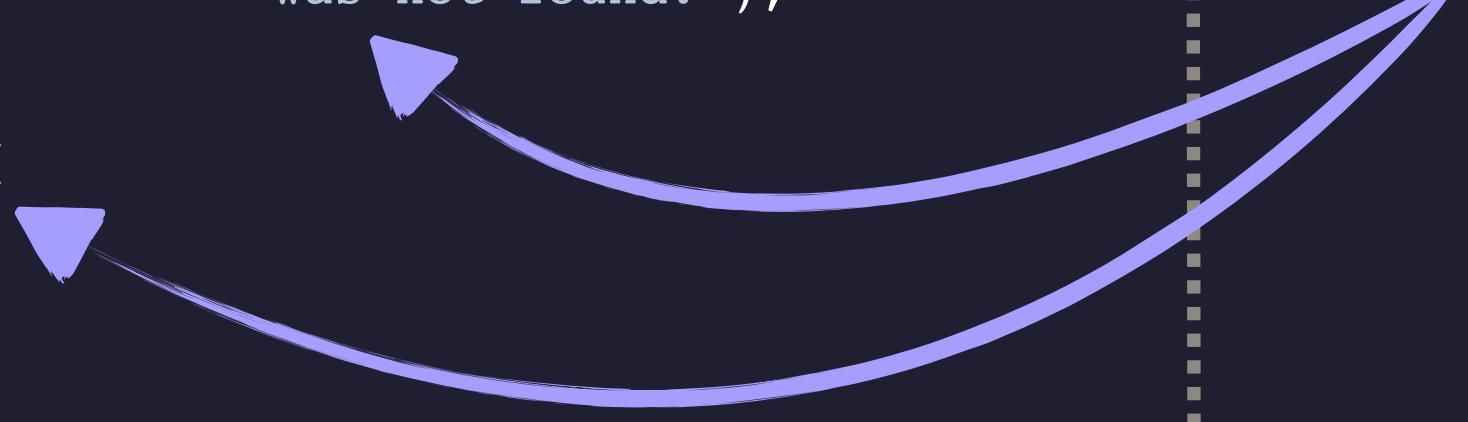


If we reach the end of the list, element location has failed.

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ) {
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i < x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
        if(i === x){
          throw new Error("An existing element" +
                        "was not found.");
        }
      }
    } catch (error2) {
    }
  }
}
```



In addition to native JS errors, we can create custom Errors with custom message properties. We'll use this message in our nested `catch`.

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i < x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
        if(i === x){
          throw new Error("An existing element" +
                         "was not found.");
        }
      }
    } catch (error2) {
      alert(error2.message + "\nCreating new text node.");
      newElement = document.createTextNode(update);
    }
  }
}
```

If no element was found with that `id` or `existing` `innerHTML`, we'll make a new text node for use somewhere outside the function.



WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i<x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
        if(i === x){
          throw new Error("An existing element" +
                         "was not found.");
        }
      }
    } catch (error2) {
      alert(error2.message + "\nCreating new text node.");
      newElement = document.createTextNode(update);
    }
  }
  finally {
    if(newElement !== undefined){
      console.log("Returning new" +
                  "text node...");  

      return newElement;
    }
  }
}
```



Then `finally`, if we made a new text node, we'll log that event out and return the new node for use.

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){
  try{
    var newElement = undefined;
    document.getElementById(id).innerHTML = update;
  } catch (error){
    try{
      var elements = document.getElementsByTagName('*');
      for(var i = 0, x = elements.length; i < x; i++){
        if(elements[i].innerHTML === existing){
          elements[i].innerHTML = update;
          id = elements[i].id;
          break;
        }
        if(i === x){
          throw new Error("An existing element" +
                         "was not found.");
        }
      }
    } catch (error2) {
      alert(error2.message + "\nCreating new text node.");
      newElement = document.createTextNode(update);
    }
  }
}
```

```
  finally {
    if(newElement !== undefined){
      console.log("Returning new" +
                  "text node...");  

      return newElement;
    } else {
      console.log("Modified element \\" +  

                  (id || existing) +  

                  "\\" with inner HTML \\" +  

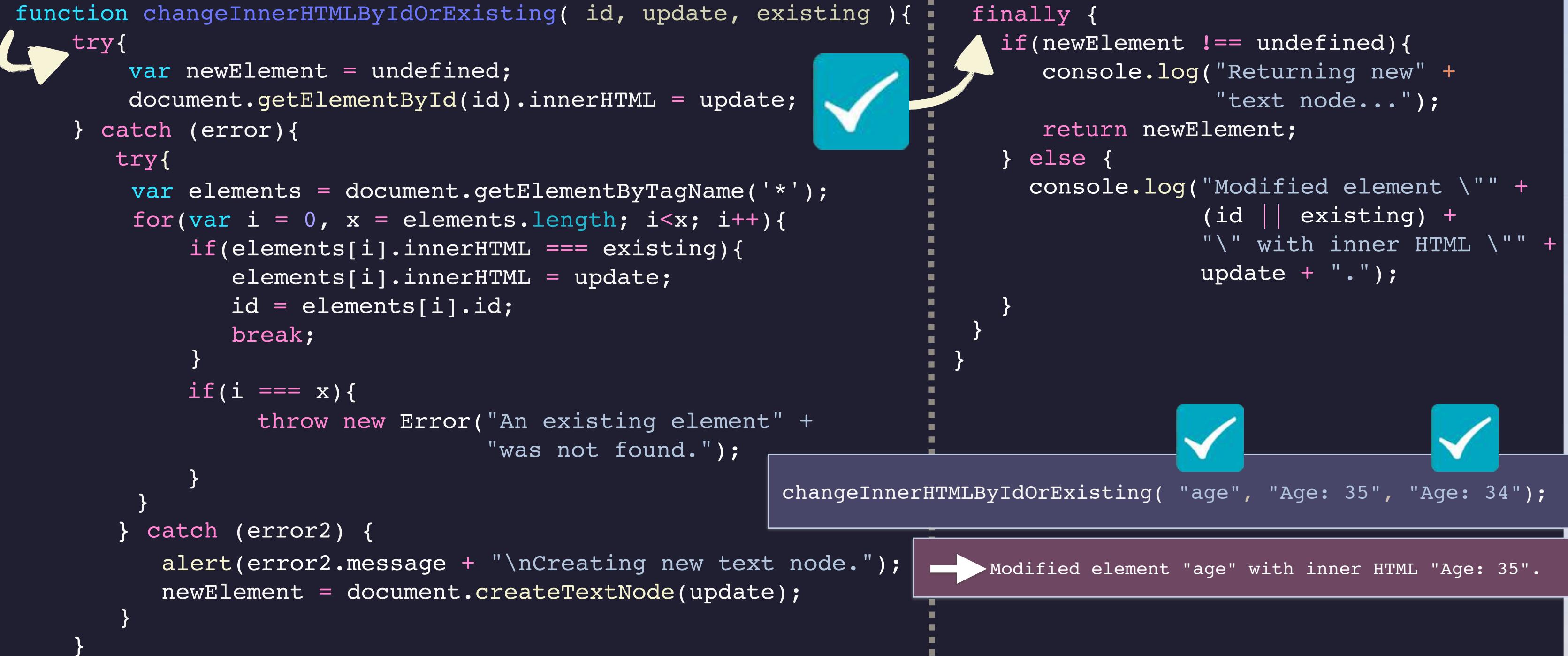
                  update + ".");
    }
  }
}
```



Otherwise, we'll notify ourselves that the requested element was updated, using either the `id` if it exists, or the `existing` HTML we found the element with.

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.



WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){  
    try{  
        var newElement = undefined;  
        document.getElementById(id).innerHTML = update;  
    } catch (error){  
        try{  
            var elements = document.getElementsByTagName('*');  
            for(var i = 0, x = elements.length; i < x; i++){  
                if(elements[i].innerHTML === existing){  
                    elements[i].innerHTML = update;  
                    id = elements[i].id;  
                    break;  
                }  
                if(i === x){  
                    throw new Error("An existing element" +  
                        "was not found.");  
                }  
            }  
        } catch (error2) {  
            alert(error2.message + "\nCreating new text node.");  
            newElement = document.createTextNode(update);  
        }  
    }  
}
```



```
    finally {  
        if(newElement !== undefined){  
            console.log("Returning new" +  
                "text node...");  
            return newElement;  
        } else {  
            console.log("Modified element \\" +  
                (id || existing +  
                "\\" with inner HTML \\" +  
                update + ".");  
        }  
    }  
}
```

No ID exists, so default selected.



```
changeInnerHTMLByIdOrExisting( "age", "Age: 35", "Age: 34");
```

→ Modified element "Age: 34" with inner HTML "Age: 35".

WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

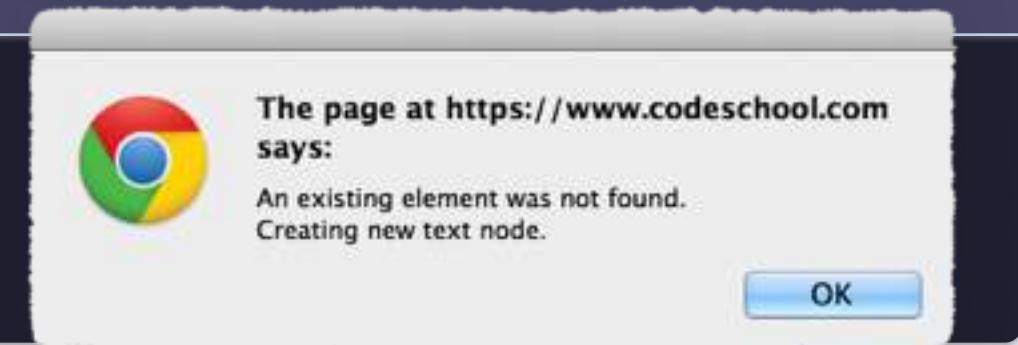
```
function changeInnerHTMLByIdOrExisting( id, update, existing ){  
    try{  
        var newElement = undefined;  
        document.getElementById(id).innerHTML = update;  
    } catch (error){  
        try{  
            var elements = document.getElementsByTagName('*');  
            for(var i = 0, x = elements.length; i < x; i++){  
                if(elements[i].innerHTML === existing){  
                    elements[i].innerHTML = update;  
                    id = elements[i].id;  
                    break;  
                }  
                if(i === x){  
                    throw new Error("An existing element" +  
                        "was not found.");  
                }  
            } catch (error2) {  
                alert(error2.message + "\nCreating new text node.");  
                newElement = document.createTextNode(update);  
            }  
        }  
    }  
}
```



```
    finally {  
        if(newElement !== undefined){  
            console.log("Returning new" +  
                "text node...");  
            return newElement;  
        } else {  
            console.log("Modified element \\" +  
                (id || existing) +  
                "\\" with inner HTML \\" +  
                update + ".");  
        }  
    }  
}
```



```
changeInnerHTMLByIdOrExisting( "age", "Age: 35", "Age: 34");
```



WHAT IF WE WANT TO TRY MORE THAN ONE OPTION?

Try-blocks nested within catch-blocks can organize an option sequence, should our first choice not work out.

```
function changeInnerHTMLByIdOrExisting( id, update, existing ){  
    try{  
        var newElement = undefined;  
        document.getElementById(id).innerHTML = update;  
    } catch (error){  
        try{  
            var elements = document.getElementsByTagName('*');  
            for(var i = 0, x = elements.length; i < x; i++){  
                if(elements[i].innerHTML === existing){  
                    elements[i].innerHTML = update;  
                    id = elements[i].id;  
                    break;  
                }  
                if(i === x){  
                    throw new Error("An existing element" +  
                        "was not found.");  
                }  
            } catch (error2) {  
                alert(error2.message + "\nCreating new text node.");  
                newElement = document.createTextNode(update);  
            }  
        }  
    }  
}
```



```
    finally {  
        if(newElement !== undefined){  
            console.log("Returning new" +  
                "text node...");  
            return newElement;  
        } else {  
            console.log("Modified element \\" +  
                (id || existing) +  
                "\\" with inner HTML \\" +  
                update + ".");  
        }  
    }  
}  
}
```



```
changeInnerHTMLByIdOrExisting( "age", "Age: 35", "Age: 34");
```



→ Returning new text node...

THE CRYSTAL OF CAUTION

Section 3
Stuff That (Sometimes) Sucks

JAVASCRIPT
BEST PRACTICES

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
};
```

```
with(drawbridge){  
}
```



The `with` keyword takes the entire encapsulated environment of the parameter object and use it to build a new “local” scope within its bracketed block ... which is kind of processing-expensive.

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrrBANG!");  
    }  
};
```

```
with(drawbridge){  
    open();  
}
```



No object name is necessary when everything in the object is treated as already local to the `with` block.



SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
};
```

```
with(drawbridge){  
}
```

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
};
```

```
with(drawbridge){  
    close = function() {  
        alert("yunyunyunyunyununCLACK!");  
    };  
}
```



Building new properties in the object
on the fly, however, doesn't work like
you might expect.

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
};
```



If a called property is not detected in the parameter object, it is considered not as a new property to be made, but rather a new global variable entirely!

```
with(drawbridge){  
    close = function() {  
        alert("yunyunyunyunyununCLACK!");  
    };  
}
```

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
};  
close = function() {  
    alert("yunyunyunyunyununCLACK!");  
};
```

Any new variables are instead added to the SURROUNDING scope...slightly counterintuitive, and not the on-the-fly property addition that we wanted.

```
with(drawbridge){  
    close = function() {  
        alert("yunyunyunyunyununCLACK!");  
    };  
}
```

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
};  
close = function() {  
    alert("yunyunyunyunyununCLACK!");  
};
```

SOMETIMES SUCKY STUFF #1: WITH

JavaScript's 'with' keyword is somewhat unreliable and often expensive, so it is generally avoided in practice.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
    close: function() {  
        alert("yunyunyunyunyununCLACK!");  
    }  
};
```



If we want any new properties within the object itself, we've still gotta do those upon construction, declaration, or with on-the-fly dot/bracket syntax.

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
var drawbridge = {  
    soldiers: 8,  
    capacity: 20,  
    open: function(){  
        alert("rrrrrrrrrrrrrBANG!");  
    }  
    close: function() {  
        alert("yunyunyunyunyununCLACK!");  
    }  
};
```

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
drawbridge  {  
  soldiers: 8,  
  capacity: 20,  
  open: function() {  
    alert("rrrrrrrrrrrrrBANG!");  
  }  
  close: function() {  
    alert("yunyunyunyunyununCLACK!");  
  }  
}
```

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
if(castle.keep.drawbridge.capacity >=  
    castle.keep.drawbridge.soldiers + reinforcements){  
    castle.keep.drawbridge.open();  
    castle.keep.drawbridge.soldiers += reinforcements;  
    alert("Drawbridge soldiers: " +  
          castle.keep.drawbridge.soldiers);  
    castle.keep.drawbridge.close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```

Needing access to the drawbridge object from the global scope would need a lot of deep access in this case.

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
if(castle.keep.drawbridge.capacity >=  
    castle.keep.drawbridge.soldiers + reinforcements){  
    castle.keep.drawbridge.open();  
    castle.keep.drawbridge.soldiers += reinforcements;  
    alert("Drawbridge soldiers: " +  
          castle.keep.drawbridge.soldiers);  
    castle.keep.drawbridge.close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
with(castle.keep.drawbridge) {  
    if(castle.keep.drawbridge.capacity >=  
        castle.keep.drawbridge.soldiers + reinforcements){  
        castle.keep.drawbridge.open();  
        castle.keep.drawbridge.soldiers += reinforcements;  
        alert("Drawbridge soldiers: " +  
              castle.keep.drawbridge.soldiers);  
        castle.keep.drawbridge.close();  
    } else {  
        alert("Reinforcements would require split unit.");  
    }  
}
```

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
with(castle.keep.drawbridge) {  
    if(  
        capacity >=  
        soldiers + reinforcements){  
        open();  
        soldiers += reinforcements;  
        alert("Drawbridge soldiers: " +  
              soldiers);  
        close();  
    } else {  
        alert("Reinforcements would require split unit.");  
    }  
}
```

WITH'S “ADMIRABLE” GOAL IS TO LIMIT REDUNDANCY

Nested objects are frequent in JavaScript, and ‘with’ attempts to help coders avoid typing deep access repeatedly.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
with(castle.keep.drawbridge) {  
    if(capacity >= soldiers + reinforcements){  
        open();  
        soldiers += reinforcements;  
        alert("Drawbridge soldiers: " + soldiers);  
        close();  
    } else {  
        alert("Reinforcements would require split unit.");  
    }  
}
```

Much more terse and legible, eh? So it makes sense to use `with` a lot, right?

WITH MAKES IT UNCLEAR WHICH PROPERTIES OR VARIABLES ARE MODIFIED

If we can't read a program and be confident in what it will do, we can't be sure that the program will work as desired.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
with(castle.keep.drawbridge) {  
    if(capacity >= soldiers + reinforcements){  
        open();  
        soldiers += reinforcements;  
        alert("Drawbridge soldiers: " + soldiers);  
        close();  
    } else {  
        alert("Reinforcements would require split unit.");  
    }  
}
```

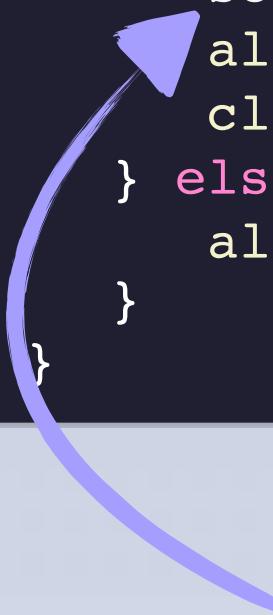
To other developers looking at this code, it would not be immediately understandable precisely where `soldiers` or `capacity` was located...within the global scope or the `with` parameter object.

WITH MAKES IT UNCLEAR WHICH PROPERTIES OR VARIABLES ARE MODIFIED

If we can't read a program and be confident in what it will do, we can't be sure that the program will work as desired.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
with(castle.keep.drawbridge) {  
    if(capacity >= soldiers + reinforcements){  
        open();  
        soldiers += reinforcements;  
        alert("Drawbridge soldiers: " + soldiers);  
        close();  
    } else {  
        alert("Reinforcements would require split unit.");  
    }  
}
```



We're not 100% sure which `soldiers` variable gets modified here. What if the `soldiers` property somehow didn't exist yet in the `drawbridge`...maybe its only gets added on the fly only in times of war!

BEST PRACTICE: USE VARIABLES TO CACHE YOUR OBJECTS

As before, caching values saves the day by letting us avoid a bunch of typing, as well as being quite clear in indication.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
with(castle.keep.drawbridge) {  
    if(capacity >= soldiers + reinforcements){  
        open();  
        soldiers += reinforcements;  
        alert("Drawbridge soldiers: " + soldiers);  
        close();  
    } else {  
        alert("Reinforcements would require split unit.");  
    }  
}
```

BEST PRACTICE: USE VARIABLES TO CACHE YOUR OBJECTS

As before, caching values saves the day by letting us avoid a bunch of typing, as well as being quite clear in indication.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
castle.keep.drawbridge  
if(capacity >= soldiers + reinforcements){  
    open();  
    soldiers += reinforcements;  
    alert("Drawbridge soldiers: " + soldiers);  
    close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```

BEST PRACTICE: USE VARIABLES TO CACHE YOUR OBJECTS

As before, caching values saves the day by letting us avoid a bunch of typing, as well as being quite clear in indication.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
var o = castle.keep.drawbridge;  
if(capacity >= soldiers + reinforcements){  
    open();  
    soldiers += reinforcements;  
    alert("Drawbridge soldiers: " + soldiers);  
    close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```



Assigning `o` to be a reference to an existing object allows us to just use that short name to get to that object.

BEST PRACTICE: USE VARIABLES TO CACHE YOUR OBJECTS

As before, caching values saves the day by letting us avoid a bunch of typing, as well as being quite clear in indication.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
var o = castle.keep.drawbridge;  
if( capacity >= soldiers + reinforcements){  
    open();  
    soldiers += reinforcements;  
    alert("Drawbridge soldiers: " + soldiers);  
    close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```

BEST PRACTICE: USE VARIABLES TO CACHE YOUR OBJECTS

As before, caching values saves the day by letting us avoid a bunch of typing, as well as being quite clear in indication.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
var o = castle.keep.drawbridge;  
if(o.capacity >= o.soldiers + reinforcements){  
    o.open();  
    o.soldiers += reinforcements;  
    alert("Drawbridge soldiers: " + o.soldiers);  
    o.close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```

BEST PRACTICE: USE VARIABLES TO CACHE YOUR OBJECTS

As before, caching values saves the day by letting us avoid a bunch of typing, as well as being quite clear in indication.

```
var castle = {  
    soldiers: 865,  
    capacity: 3000,  
    keep: {  
        soldiers: 19,  
        capacity: 40,  
        drawbridge: {  
            soldiers: 8,  
            capacity: 20,  
            open: function(){  
                alert("rrrrrrrrrrrrBANG!");  
            }  
            close: function() {  
                alert("yunyunyunyununCLACK!");  
            }  
        }  
    }  
};
```

```
var reinforcements = 12;  
var o = castle.keep.drawbridge;  
if(o.capacity >= o.soldiers + reinforcements){  
    o.open();  
    o.soldiers += reinforcements;  
    alert("Drawbridge soldiers: " + o.soldiers);  
    o.close();  
} else {  
    alert("Reinforcements would require split unit.");  
}
```



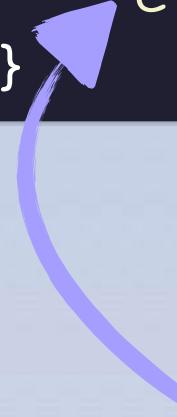
With cached clarity, team members will be able to understand the code's functionality. Also, no lengthy nested object names!

SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "'");  
}
```

The eval method will take a string as a parameter, start the JavaScript compiler, and treat that string as if it were a line of code to execute.



SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "'");  
}
```

```
assignRegimentMotto(2, "The Best Of The Best");
```

```
regiment + 2 + .motto = ' + The Best Of The Best + '
```

SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "'");  
}
```

```
assignRegimentMotto(2, "The Best Of The Best");
```

```
regiment 2 .motto = ' The Best Of The Best '
```

SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "'");  
}
```

```
assignRegimentMotto(2, "The Best Of The Best");
```

```
regiment2.motto = 'The Best Of The Best'
```

SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "'");  
}
```

```
assignRegimentMotto( , );
```

```
regiment .motto = '
```

SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "');  
}
```

```
assignRegimentMotto(1, "The King's Own");
```

regiment1.motto = 'The King's Own'



The resulting code will be invalid because of the extra apostrophe, which makes the compiler think the string is complete...until it sees more stuff and then marks the code as invalid JS.

SOMETIMES SUCKY STUFF #2: EVAL

JavaScript's 'eval' keyword may not be evil, but it can affect legibility, an ability to debug, and performance.

```
function assignRegimentMotto(number, motto) {  
    eval("regiment" + number + ".motto = '" + motto + "'");  
}
```

```
assignRegimentMotto(1, "The King's Own");
```

regiment1.motto = 'The King's Own'



Additionally, we may not have any idea what broke our code, if the data is ever passed by variable instead of by value. Was it `number`? Was it `motto`? Something else?

IF EVAL IS A MUST FOR SOME REASON, EVAL AS LITTLE AS POSSIBLE

Eval might be useful for dynamic code or uncontrollable data, but it's still treating the string as a program to expensively compile.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number + ".motto = '" + motto + "');  
}
```

```
assignRegimentMotto(1, "The King's Own");
```

IF EVAL IS A MUST FOR SOME REASON, EVAL AS LITTLE AS POSSIBLE

Eval might be useful for dynamic code or uncontrollable data, but it's still treating the string as a program to expensively compile.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number  
}
```

```
assignRegimentMotto(1, "The King's Own");
```

IF EVAL IS A MUST FOR SOME REASON, EVAL AS LITTLE AS POSSIBLE

Eval might be useful for dynamic code or uncontrollable data, but it's still treating the string as a program to expensively compile.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number).motto = motto;  
}
```

```
assignRegimentMotto(1, "The King's Own");
```

Try to minimize the operations that your new mini-program needs to engage in ... which will also have the benefit of improving your debug capability.

IF EVAL IS A MUST FOR SOME REASON, EVAL AS LITTLE AS POSSIBLE

Eval might be useful for dynamic code or uncontrollable data, but it's still treating the string as a program to expensively compile.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number).motto = motto;  
}
```

```
assignRegimentMotto(1, "The King's Own");  
console.log(regiment1.motto);
```

→ The King's Own

Try to minimize the operations that your new mini-program needs to engage in ... which will also have the benefit of improving your debug capability.

BEST PRACTICE: FOR BASIC MAPPINGS, USE YOUR DATA STRUCTURES

Eval is most often misused for just this kind of mapping numbers to objects, but an array is much more efficient.

```
function assignRegimentMotto(number, motto){  
    eval("regiment" + number).motto = motto;  
}
```

BEST PRACTICE: FOR BASIC MAPPINGS, USE YOUR DATA STRUCTURES

Eval is most often misused for just this kind of mapping numbers to objects, but an array is much more efficient.

```
function assignRegimentMotto(number, motto){  
}
```

BEST PRACTICE: FOR BASIC MAPPINGS, USE YOUR DATA STRUCTURES

Eval is most often misused for just this kind of mapping numbers to objects, but an array is much more efficient.

```
var regiments = [**a bunch of ordered regiment objects**];  
function assignRegimentMotto(number, motto){  
}  
  
// Example:  
// regiments[0].motto = "The King's Own"  
// regiments[1].motto = "The Royal Scots"  
// regiments[2].motto = "The Royal Ulster Rifles"
```



No need to number the variable name if your objects are already in order. Build your own data with its usage in mind!

BEST PRACTICE: FOR BASIC MAPPINGS, USE YOUR DATA STRUCTURES

Eval is most often misused for just this kind of mapping numbers to objects, but an array is much more efficient.

```
var regiments = [**a bunch of ordered regiment objects**];  
function assignRegimentMotto(number, motto){  
    regiments[number].motto = motto;  
}
```



Instead of `eval`, use the associative nature of arrays to build any dynamic mapping that you desire. Referencing an array is much faster than parsing and running a new mini-program.

BUT, BUT, WHAT IF I NEED TO READ AND WRITE STRINGY JSON DATA?

Chill, bro. Using eval to parse JSON data is often regarded as vulnerable and unsafe, because eval evaluates any script.

```
var regimentsJSON = '{' +  
  '"regiment1": { "motto": "The King\\'s Own", ' +  
    ' "numMembers": 46, ' +  
    ' "captain": "Jar Treen" }, ' +  
  '"regiment2": { "motto": "The Best of the Best", ' +  
    ' "numMembers": 40, ' +  
    ' "captain": "Maximo Rarter" }, ' +  
  ' ...  
' }';
```

Some stringify'd JSON data.

```
var regiments = eval(regimentsJSON);
```

If this data came from somewhere else
(as does a lot of JSON data), there could
be a potential vulnerability to malicious
scripts through “code injection.”

BUT, BUT, WHAT IF I NEED TO READ AND WRITE STRINGY JSON DATA?

Chill, bro. Using eval to parse JSON data is often regarded as vulnerable and unsafe, because eval evaluates any script.

```
var regimentsJSON = '{' +  
  '"regiment1": { "motto": "The King\\'s Own", ' +  
    ' "numMembers": 46, ' +  
    ' "captain": "Jar Treen" }, ' +  
  '"regiment2": { "motto": "The Best of the Best", ' +  
    ' "numMembers": 40, ' +  
    ' "captain": "Maximo Rarter" }, ' +  
  ...  
' }';
```

```
var regiments = eval(regimentsJSON);
```

USE JSONPARSE() TO ENSURE ONLY JSON IS ACCEPTED

JSON.parse, or a parser library that recognizes JSON, helps to avoid both the security and performance issues posed by eval.

```
var regimentsJSON = '{' +  
  '"regiment1": { "motto": "The King\'s Own", ' +  
    ' "numMembers": 46, ' +  
    ' "captain": "Jar Treen" }, ' +  
  '"regiment2": { "motto": "The Best of the Best", ' +  
    ' "numMembers": 40, ' +  
    ' "captain": "Maximo Rarter" }, ' +  
  ...  
' }';
```

```
var regiments = JSON.parse(regimentsJSON);
```

```
regiments  
  regiment1: {  
    motto: "The King's Own",  
    numMembers: 46,  
    captain: "Jar Treen"  
  },  
  regiment2: {  
    motto: "The Best of the Best",  
    numMembers: 40,  
    captain: "Maximo Rarter"  
  },  
  ...
```

SOMETIMES SUCKY STUFF #3: LEAVING OFF { }

Just because we CAN leave curly braces off of single-statement blocks of code, doesn't mean we should.

```
if(isKing)
    weapon = "Excalibur";
else
    weapon = "Longsword";
```

When a conditional or loop has only one statement to execute, it's true that no curly brackets are needed to enclose the block.

Same goes for the else portion.

BUT WHAT IF NEW CODE NEEDS TO BE ADDED?

Requiring bracket analysis in order to ensure the proper scoping of necessary additional code will ensure your unpopularity.

```
if(isKing)
    weapon = "Excalibur";
else
    weapon = "Longsword";
```

BUT WHAT IF NEW CODE NEEDS TO BE ADDED?

Requiring bracket analysis in order to ensure the proper scoping of necessary additional code will ensure your unpopularity.

```
if(isKing)
    weapon = "Excalibur";
    alert("Hail Arthur, King of the Britons!");
    removeFromArmory("Excalibur");
else
    weapon = "Longsword";
    alert("Charge on, Sir Knight!");
    removeFromArmory("Longsword");
```

Now this `else` will cause an error, since an immediately preceding `if`-block is not found. This may not even be discovered until some function is called...



MISSING BRACKETS DO NOT ALWAYS GENERATE ERRORS

In the example of conditional blocks, leaving off brackets may produce valid JavaScript and undesired executions!

```
if(isKing)
    weapon = "Excalibur";
    alert("Hail Arthur, King of the Britons!");
    removeFromArmory("Excalibur");
else
    weapon = "Longsword";
    alert("Charge on, Sir Knight!");
    removeFromArmory("Longsword");
```

MISSING BRACKETS DO NOT ALWAYS GENERATE ERRORS

In the example of conditional blocks, leaving off brackets may produce valid JavaScript and undesired executions!

```
if(isKing)
    weapon = "Excalibur";
    alert("Hail Arthur, King of the Britons!");
removeFromArmory("Excalibur");
```

MISSING BRACKETS DO NOT ALWAYS GENERATE ERRORS

In the example of conditional blocks, leaving off brackets may produce valid JavaScript and undesired executions!

```
if(isKing)
  weapon = "Excalibur";
  alert("Hail Arthur, King of the Britons!");
removeFromArmory("Excalibur");
```

Now, even if `isKing` is false, the `alert` and the call to `removeFromArmory` will both execute.

MISSING BRACKETS DO NOT ALWAYS GENERATE ERRORS

In the example of conditional blocks, leaving off brackets may produce valid JavaScript and undesired executions!

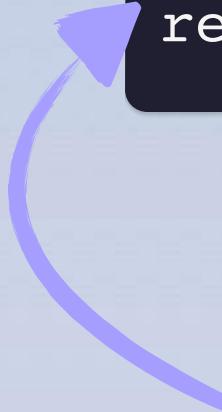
```
if(isKing)
    weapon = "Excalibur";
    alert("Hail Arthur, King of the Britons!");
removeFromArmory("Excalibur");
```

MISSING BRACKETS DO NOT ALWAYS GENERATE ERRORS

In the example of conditional blocks, leaving off brackets may produce valid JavaScript and undesired executions!

```
if(isKing)
    weapon = "Excalibur";
alert("Hail Arthur, King of the Britons!");
removeFromArmory("Excalibur");
```

Without the `else` to trigger an error,
the JS interpreter just “sees” the
code written like this.



MISSING BRACKETS DO NOT ALWAYS GENERATE ERRORS

In the example of conditional blocks, leaving off brackets may produce valid JavaScript and undesired executions!

```
if(isKing)
    weapon = "Excalibur";
alert("Hail Arthur, King of the Britons!");
removeFromArmory("Excalibur");
```

BE COOL. BRACKET YOUR CODE.

For better organization, legibility, and error prevention, brackets are a two-character sacrifice that you should be making.

```
if(isKing) {  
    weapon = "Excalibur";  
    alert("Hail Arthur, King of the Britons!");  
    removeFromArmory("Excalibur");  
} else {  
    weapon = "Longsword";  
    alert("Charge on, Sir Knight!");  
    removeFromArmory("Longsword");  
}
```



THE CRYSTAL OF CAUTION

Section 4
Number Nonsense

JAVASCRIPT
BEST PRACTICES

DECIMALS IN JAVASCRIPT ARE WACKY

JavaScript uses binary floating point values to handle all of its decimal based operations.

```
console.log(0.1 + 0.2);
```

→ 0.3000000000000004



Um.

DECIMALS IN JAVASCRIPT ARE WACKY

JavaScript uses binary floating point values to handle all of its decimal based operations.

```
console.log(0.1 + 0.2);
```

→ 0.3000000000000004

```
console.log(0.1 + 0.2 + 0.3);
```

→ 0.6000000000000001

```
console.log((0.1 + 0.2) + 0.3);
```

→ 0.6000000000000001

```
console.log(0.1 + (0.2 + 0.3));
```

→ 0.6

Cool, so I'll just jump off
a bridge and everything
will be fine.

A FEW METHODS HELP IMPROVE DECIMAL VISUALIZATION

First, the `toFixed()` method will allow you to select the exact amount of decimal places to display.

```
console.log(0.1 + 0.2);
```

→ 0.3000000000000004

```
console.log(0.1 + 0.2 + 0.3);
```

→ 0.6000000000000001

```
var num = 0.1 + 0.2;  
console.log(num.toFixed(1));
```

→ 0.3

```
var num = 0.1 + 0.2 + 0.3;  
console.log(num.toFixed(4));
```

→ 0.6000

toFixed() WILL ROUND TO THE LAST INDICATED POSITION

If dealing with percentages of money, for example, `toFixed()` can handle hundredths rounding for you.

```
function tax (price, percent){  
    return (price*percent/100).toFixed(2);  
}
```

```
tax(9.85, 7.5);
```

→ "0.74"

2 decimal places,
but wait ... a string?

$$9.85 * 7.5 / 100 = .73875$$



The `toFixed(2)` looks at the third place to ensure two places will be correctly rounded.

toFixed() WILL ROUND TO THE LAST INDICATED POSITION

If dealing with percentages of money, for example, `toFixed()` can handle hundredths rounding for you.

```
function tax (price, percent){  
    return (price*percent/100).toFixed(2);  
}
```

```
tax(9.85, 7.5);
```

→ "0.74"

```
var mailedGlove = 9.85;  
var armorTax = 7.5;  
var total = mailedGlove + tax(mailedGlove, armorTax);
```

→ "9.850.74"

Well that escalated quickly.
We'll need a new method to
help use the rounded value in
further math operations.



PARSEFLOAT() TURNS STRINGS WITH DECIMALS INTO NUMBERS

A combination of `toFixed()` and `parseFloat()` will let us use values of exact length in other math operations.

```
function tax (price, percent){  
    return (price*percent/100).toFixed(2);  
}
```

PARSEFLOAT() TURNS STRINGS WITH DECIMALS INTO NUMBERS

A combination of `toFixed()` and `parseFloat()` will let us use values of exact length in other math operations.

```
function tax (price, percent){  
    return (price*percent/100).toFixed(2);  
}
```

PARSEFLOAT() TURNS STRINGS WITH DECIMALS INTO NUMBERS

A combination of `toFixed()` and `parseFloat()` will let us use values of exact length in other math operations.

```
function tax (price, percent){  
    return parseFloat((price*percent/100).toFixed(2));  
}
```

```
var mailedGlove = 9.85;  
var armorTax = 7.5;  
var total = mailedGlove + tax(mailedGlove, armorTax);  
console.log(total);
```

→ 10.59



Good to go for either display usage or further math operations.

PARSEINT() WILL ALSO CONVERT NUMERICAL STRINGS

Instead of looking for a floating point number, `parseInt()` seeks the first available integer at the front of a string.

```
parseInt("88");
```

→ 88

```
parseInt("88 keys on a piano");
```



→ 88

`parseInt()` will try to help out and provide any integer that begins a string of other characters ... slightly weird ...and maybe dangerous.

```
parseFloat("3.28084 meters in a foot");
```



→ 3.28084

Same thing with `parseFloat()` too ... but it looks for the termination of numbers after a decimal.

PARSEINT() WILL ALSO CONVERT NUMERICAL STRINGS

Instead of looking for a floating point number, `parseInt()` seeks the first available integer at the front of a string.

```
parseInt("88");
```

→ 88

```
parseInt("88 keys on a piano");
```



→ 88

`parseInt()` will try to help out and provide any integer that begins a string of other characters ... slightly weird ...and maybe dangerous.

```
parseInt("There are 88 keys on a piano");
```



→ NaN

If a string does not begin with an acceptable value for either method, however, you'll get the JS value `NaN`, or “Not a Number.” More on this in a bit.

PARSEINT() WILL ALSO CONVERT NUMERICAL STRINGS

Instead of looking for a floating point number, `parselnt()` seeks the first available integer at the front of a string.

```
parseInt("88");
```



```
parseInt("88 keys on a piano");
```



```
parseInt("There are 88 keys on a piano");
```



PARSEINT() WILL ALSO CONVERT NUMERICAL STRINGS

Instead of looking for a floating point number, `parseInt()` seeks the first available integer at the front of a string.

```
parseInt("88");
```

→ 88

```
parseInt("88 keys on a piano");
```

→ 88

```
parseInt("9.85");
```

→ 9



```
parseInt("There are 88 keys on a piano");
```

→ NaN

A `parseInt()` will also trim off any decimals that may exist, without rounding.

PARSEINT() CAN CAUSE UNEXPECTED VALUE ERRORS

Since parseInt() will accept octal, hexadecimal, and decimal values, caution is necessary.

```
var userAge = "021";
```



Say that a user's age got assigned to a variable as a string from a form entry... but the user made an error in typing.

PARSEINT() CAN CAUSE UNEXPECTED VALUE ERRORS

Since parseInt() will accept octal, hexadecimal, and decimal values, caution is necessary.

```
var userAge = "021";
```

```
parseInt(userAge);
```

→ 21
↑

ECMAScript 5 systems will provide exactly what you might expect, in the event of a mis-entered zero ...

```
parseInt(userAge);
```

→ 17
↑

... older systems think you are trying to pass an octal value (base 8), and try to help you out with a conversion back to decimal.

USE A RADIX VALUE TO ENSURE CORRECT PARSING

When a browser version or user input is not reliable, make sure to let `parseInt()` what number system you expect.

```
var userAge = "021";
```

```
parseInt(userAge);
```

→ 21

```
parseInt(userAge);
```

→ 17

```
parseInt("021", 10);
```

→ 21 

`parseInt()` will accept any radix value from 2-36 upon which to base the numerical string it has received. Do this for the rest of your natural born life.

TESTING FOR NUMBER PRESENCE BEFORE OPERATIONS

Using NaN to check if values are actually JS Numbers seems like a good idea, but it will try to kill you.

```
typeof NaN;
```

→ "number"



TESTING FOR NUMBER PRESENCE BEFORE OPERATIONS

Using NaN to check if values are actually JS Numbers seems like a good idea, but it will try to kill you.

```
typeof NaN;
```

→ "number"

```
console.log(NaN === NaN);
```

→ false



TESTING FOR NUMBER PRESENCE BEFORE OPERATIONS

Using NaN to check if values are actually JS Numbers seems like a good idea, but it will try to kill you.

```
typeof NaN;
```

→ "number"

```
console.log(NaN === NaN);
```

→ false

```
isNaN("42");
```

→ false

This ... is a string. This is a string, and not a number. It is not a number, because it is ... a string.



DO A DOUBLE CHECK TO AVOID INSANITY

If unsure about data type, but highly reliant on a Number, use `typeof` AND `isNaN()` as a best practice.

```
typeof NaN;
```

→ "number"

```
console.log(NaN === NaN);
```

→ false

```
isNaN("42");
```

→ false

Turns out that this method is looking strictly for the JS value `Nan` itself. Fun.

DO A DOUBLE CHECK TO AVOID INSANITY

If unsure about data type, but highly reliant on a Number, use `typeof` AND `isNaN()` as a best practice.

```
typeof NaN;
```

→ "number"

```
console.log(NaN === NaN);
```

→ false

```
isNaN("42");
```

→ false

```
isThisActuallyANumberDontLie(640);
```

→ true

```
isThisActuallyANumberDontLie("640");
```

→ false

```
isThisActuallyANumberDontLie(NaN);
```

→ false

```
function isThisActuallyANumberDontLie( data ){
  return ( typeof data === "number" && !isNaN(data) );
}
```

DO A DOUBLE CHECK TO AVOID INSANITY

If unsure about data type, but highly reliant on a Number, use `typeof` AND `isNaN()` as a best practice.

```
function isThisActuallyANumberDontLie( data ){
  return ( typeof data === "number" && !isNaN(data) );
}
```

USE PARSE METHODS FOR LENIENCY OR FORM DATA

If numerical data may show up as a string, parse data before your type check and add additional checks.

```
function isThisActuallyANumberDontLie( data ){
    return ( typeof data === "number" && !isNaN(data) );
}
```

Zip Code: 32803

true

Zip Code: 3280#

false



The page at <https://www.codeschool.com> says:

Please enter a valid zip, dude.

OK

```
function checkValidZip( ){
    var entry = document.getElementById("zip").value;
    var userZip = parseInt(entry);
    try{
        if(isThisActuallyANumberDontLie( userZip ) ){
            if(userZip.toFixed(0).length === 5) {
                return true;
            } else {
                throw new Error("Nope!");
            }
        } else{
            throw new Error("Nope!");
        }
    } catch(e){
        if(e.message === "Nope!"){
            alert("Please enter a valid zip, dude.");
            return false;
        }
        // other error responses go here...
    }
}
```

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

```
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"],
  nor = 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

```
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.

```
<!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;

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);
}
```

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.

```
<!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);
}
```

ENTER THE SIMULATED JAVASCRIPT NAMESPACE

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

```
<!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

ENTER THE SIMULATED JAVASCRIPT NAMESPACE

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

```
<!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 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.

```
<!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 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.

```
<!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 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`.

HALLOFFAME.JS

```
<!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>

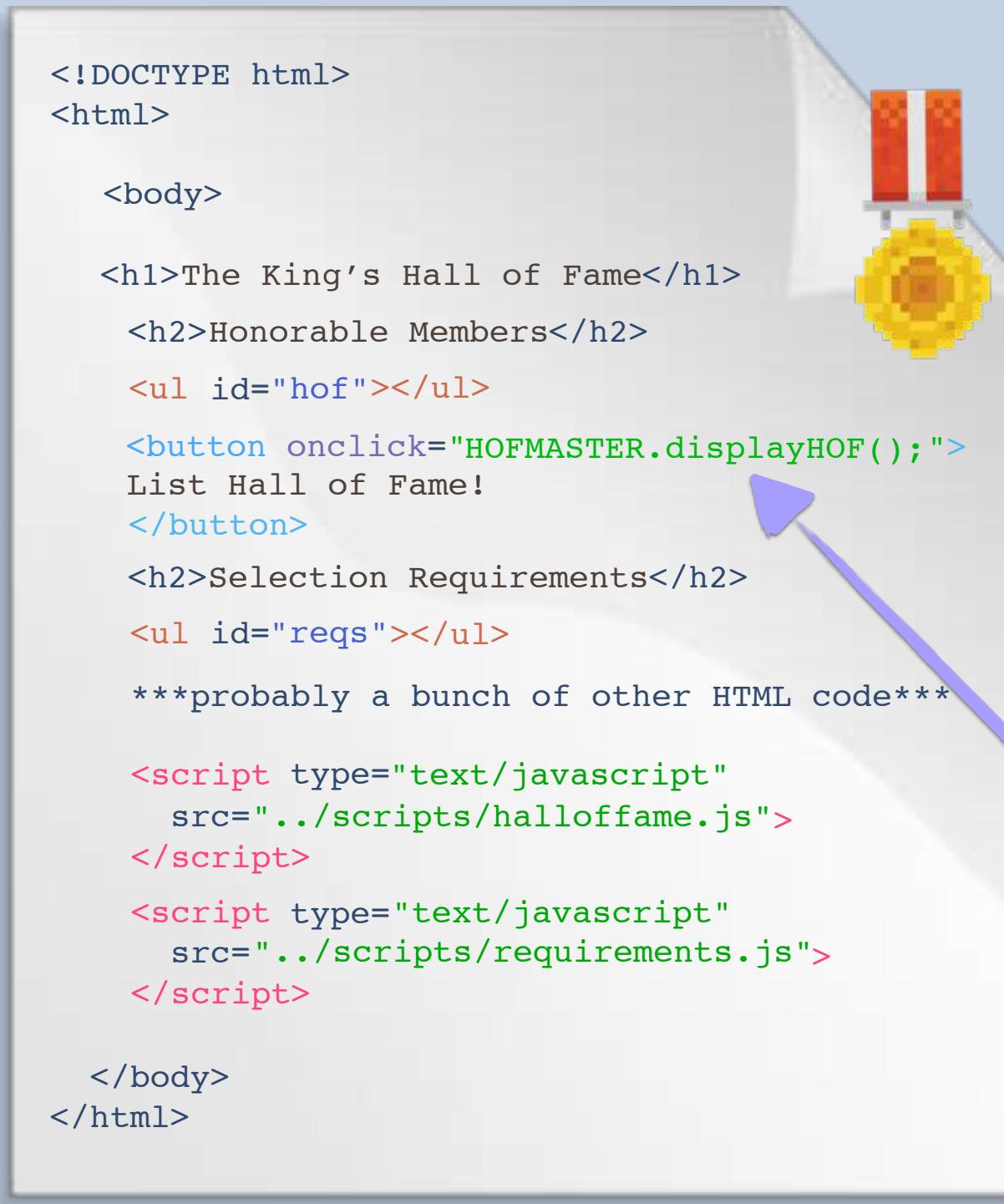
  </body>
</html>
```



```
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

```
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.

```
<!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);
  }
};
```

NESTED NAMESPACING IS FREQUENT IN A MODULE PATTERN

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

```
<!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);
  },
  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 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(...){}  
  
};
```

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 makes 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 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.

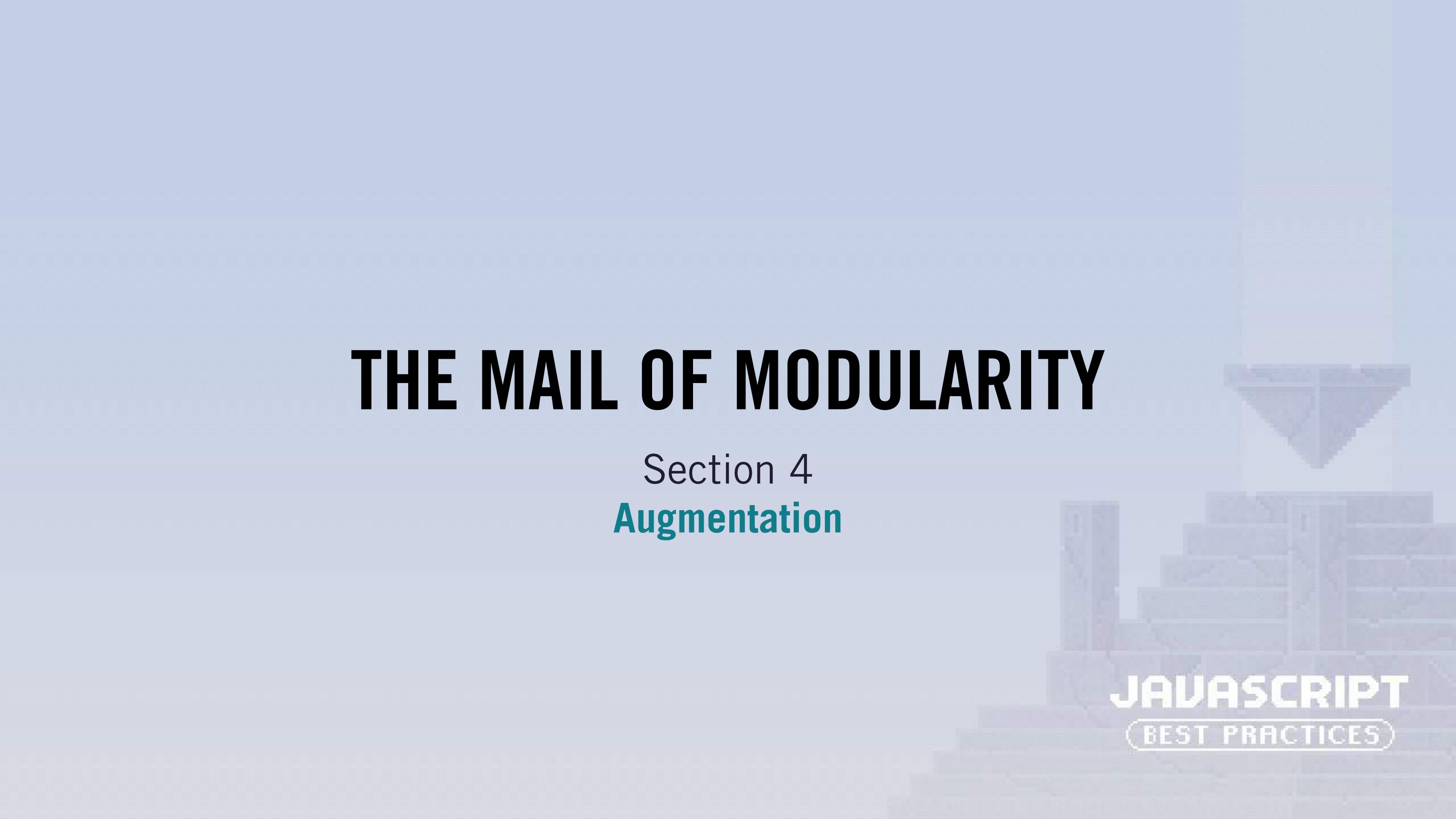
```
var wartime = true;
var ARMORY = (function( war ){
    var weaponList = [ *list of weapon Objects* ];
    var armorList = [ *list of armor Objects* ];
    var removeWeapon = function(...){};
    var replaceWeapon = function(...){};
    var removeArmor = function(...){};
    var replaceArmor = function(...){};

    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 4
Augmentation



JAVASCRIPT
BEST PRACTICES

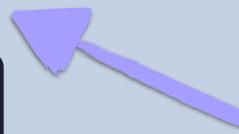
MODULES OFTEN NEED TO HAVE ADDITIONS TO THEIR EXISTING PROPERTIES

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(...){};

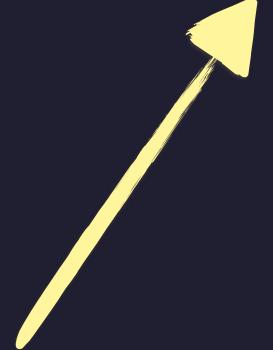
  return {
    makeWeaponRequest: function(...){
      if(war) //let civilians have weaponry
    },
    makeArmorRequest: function(){}
  };

})(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 ){
  ...
})( ARMORY );
```

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



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.