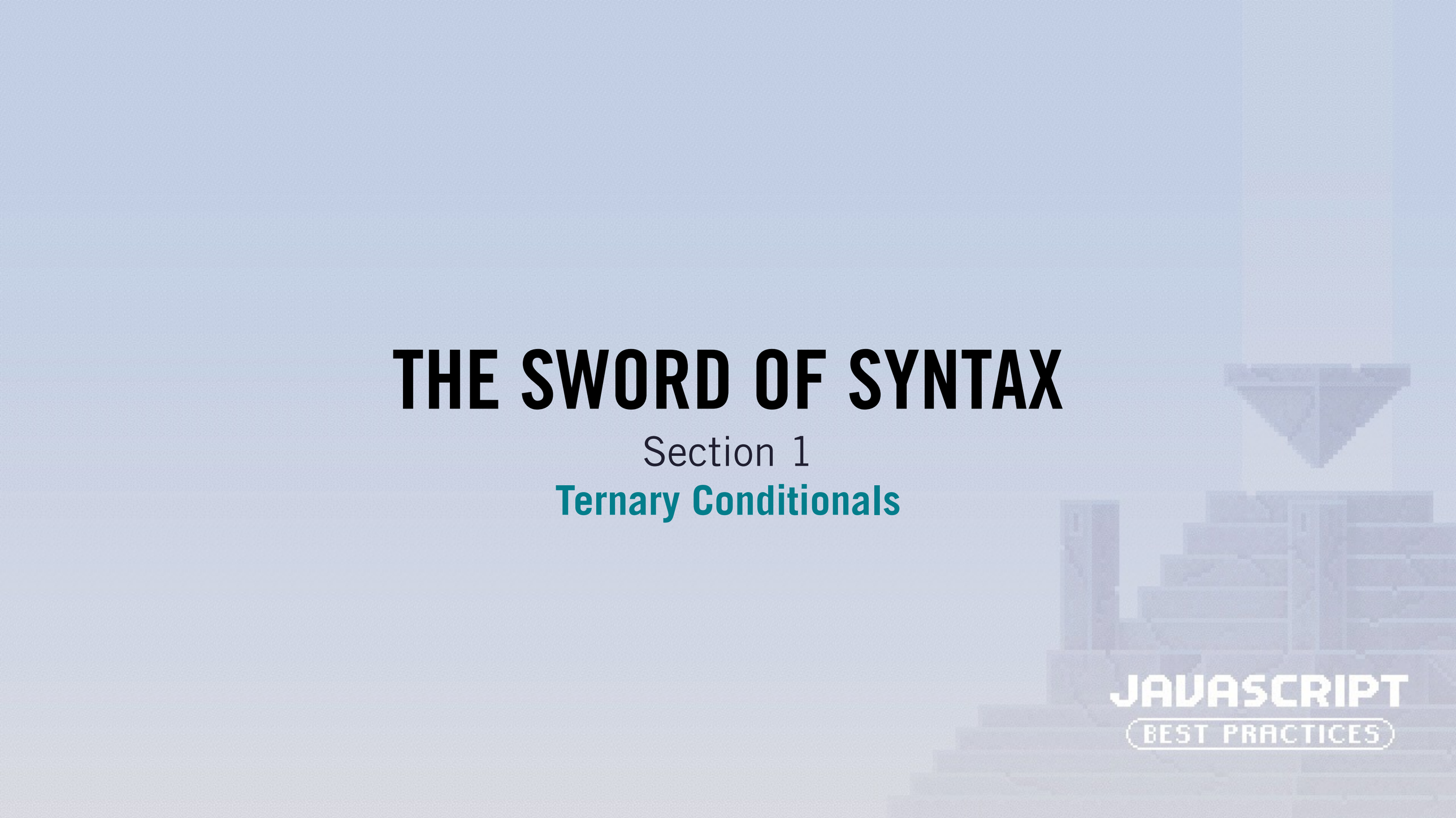
THE SWORD OF SYNTAX

Section 1

Ternary Conditionals



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";
}
```

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

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

→ "Longsword"

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

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";
}
```

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

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

→ "Longsword"

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

"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";
}
```

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

```
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";
}
```

```
                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");
```

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";
}
```

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

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

"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");
```

❌ → 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";
}
```

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".

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

❌ → Excalibur

"Current weapon: false" ? "Excalibur" : "Longsword"

String

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.

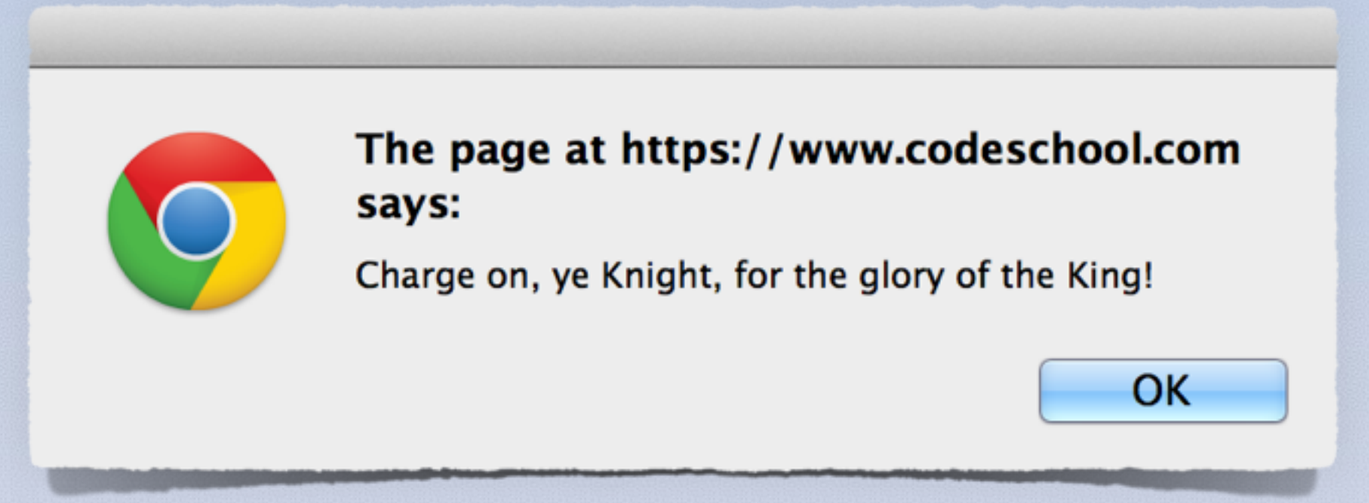
```
var isArthur = true;
var isKing = true;
```


```
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;
```




```
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.

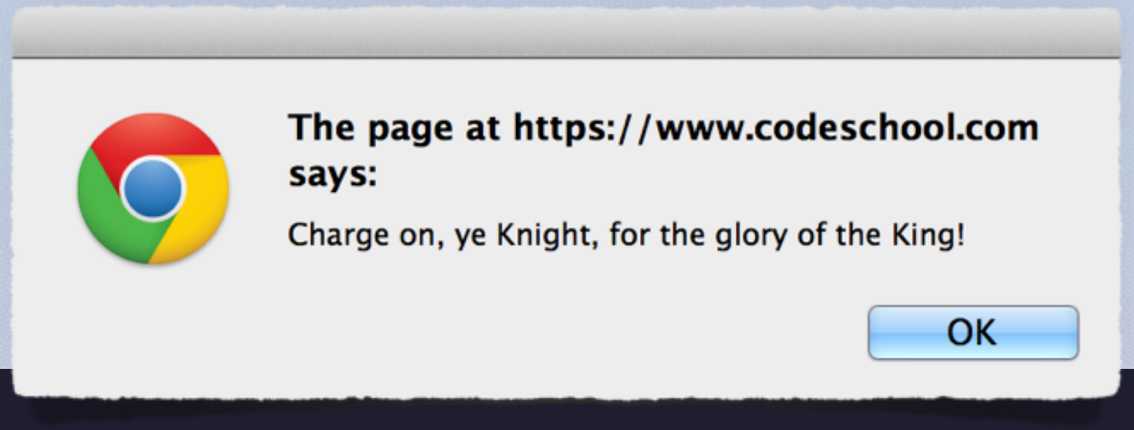
```
var isArthur = true;
var isKing = false;
```

```
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.

→ `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
```

SYNTAX

Section 1

Ternary Conditionals

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    [ ];
                    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 = [ … ];
```

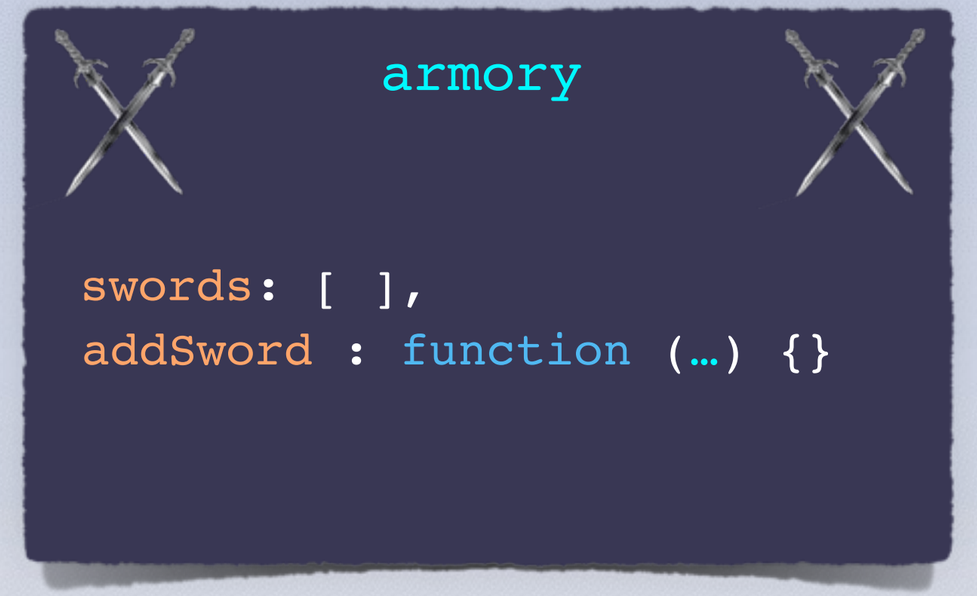
As soon as something "not false" is found, it is accepted for the assignment. In this case, our `[ ]` never even gets looked at. This event is often called "short-circuiting."

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");
```

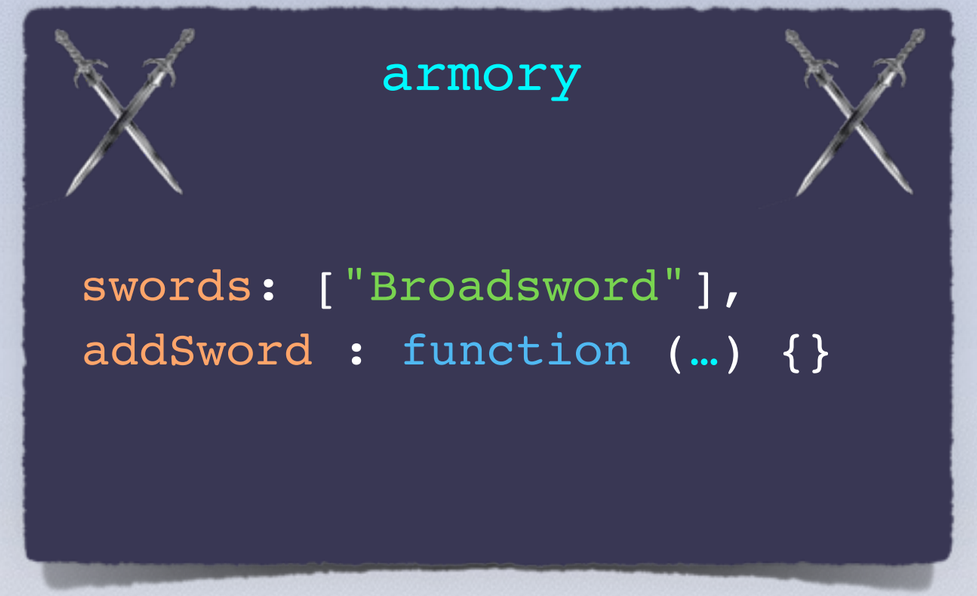


ASSIGNMENTS WITH LOGICAL OPERATORS

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");
```

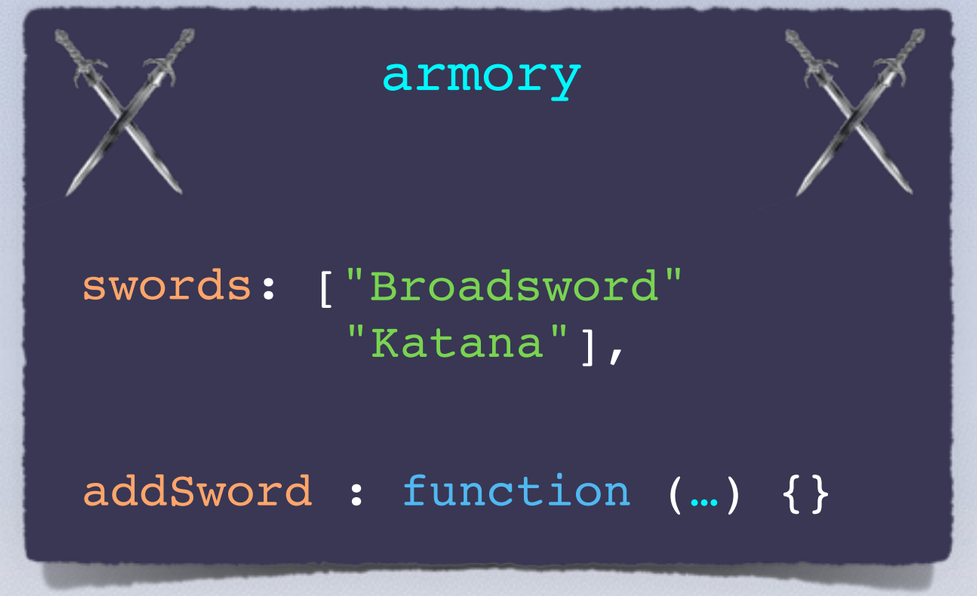


ASSIGNMENTS WITH LOGICAL OPERATORS

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");
```

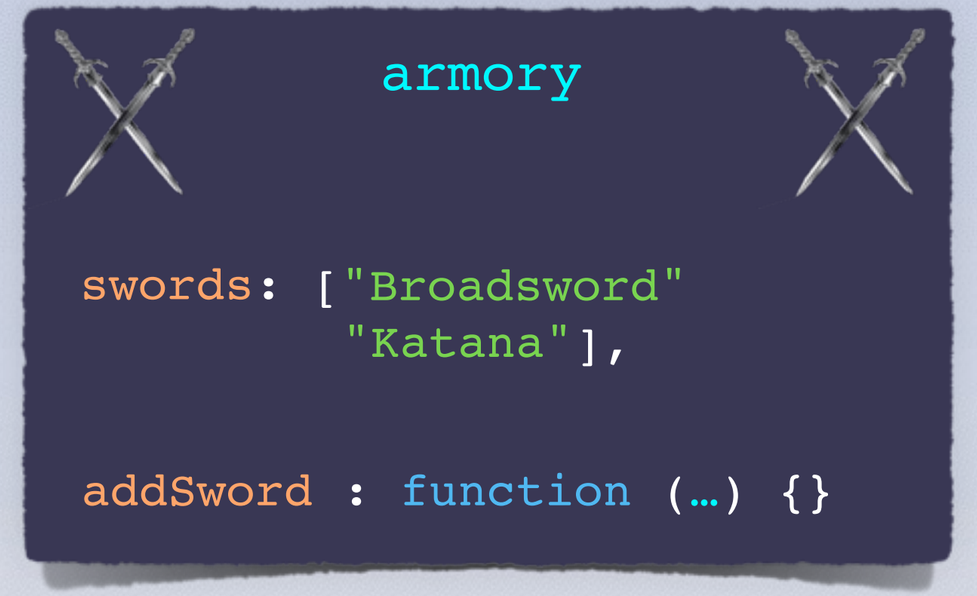


ASSIGNMENTS WITH LOGICAL OPERATORS

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");
```

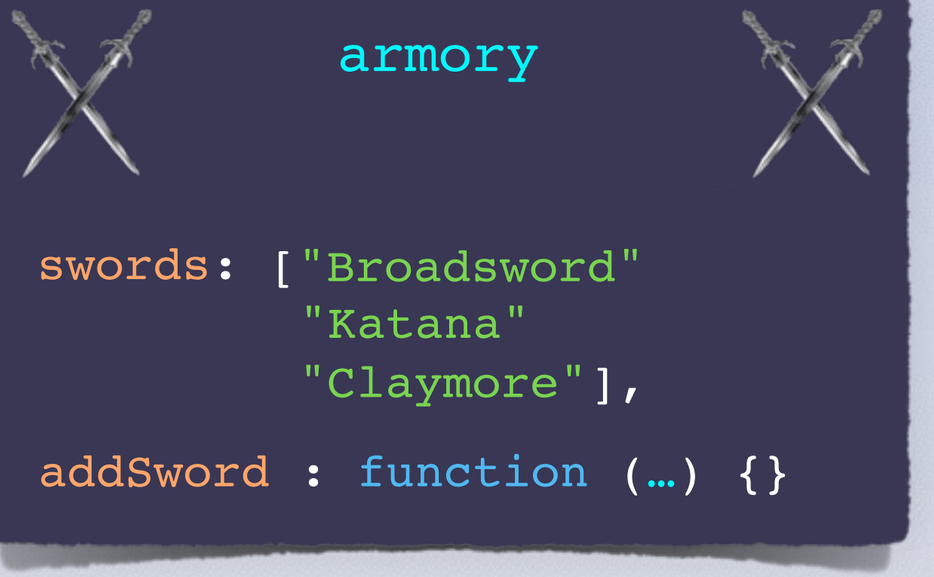


ASSIGNMENTS WITH LOGICAL OPERATORS

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");
```



ASSIGNMENTS WITH LOGICAL OPERATORS

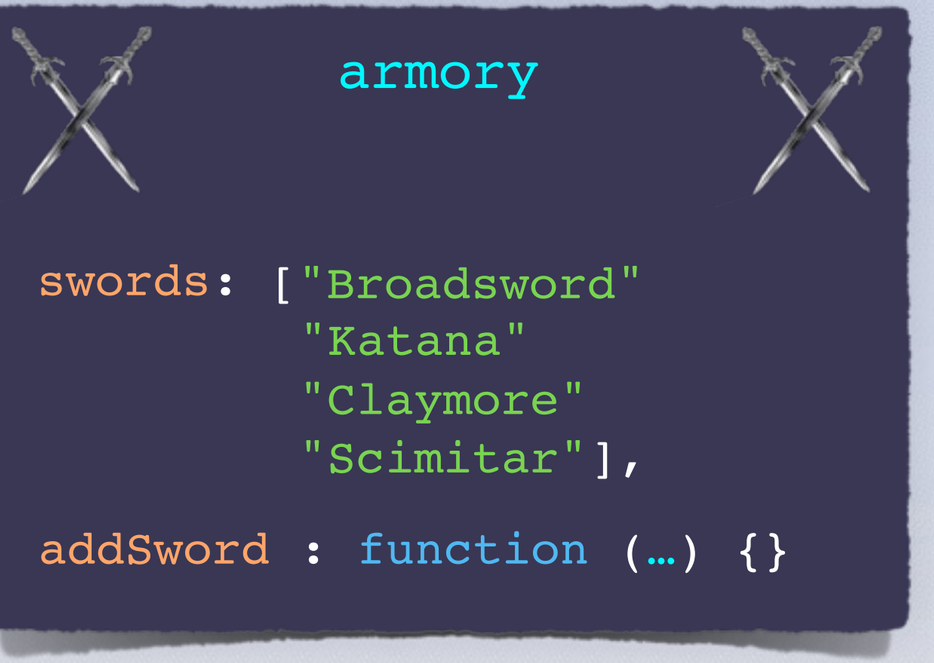
Using a logical assignment here will eliminate a read of the sword property.

```
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);
```

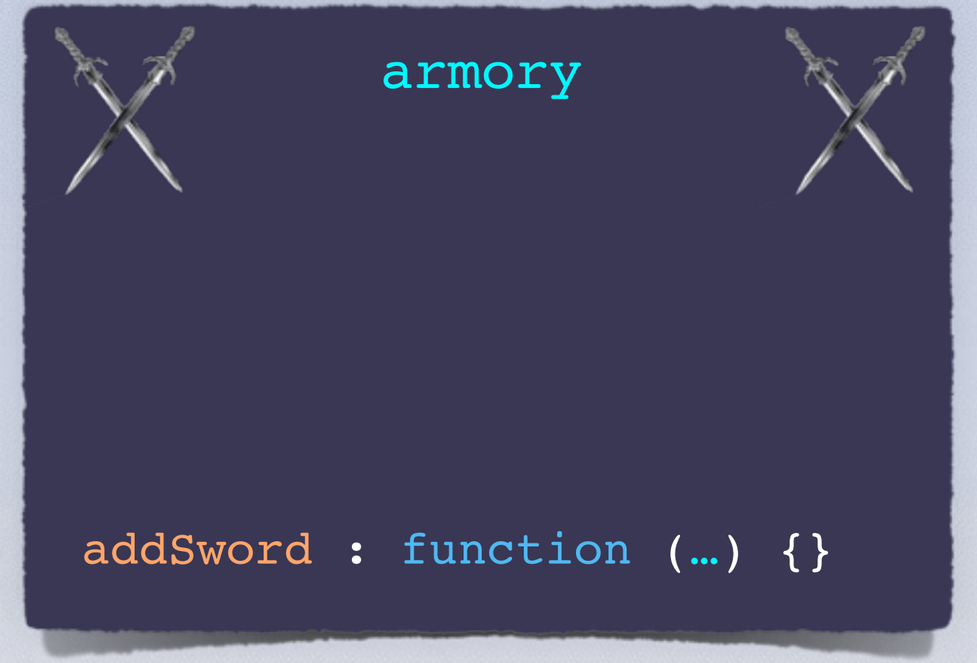
→ ["Broadsword", "Katana", "Claymore", "Scimitar"]



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);
                }
            };
```



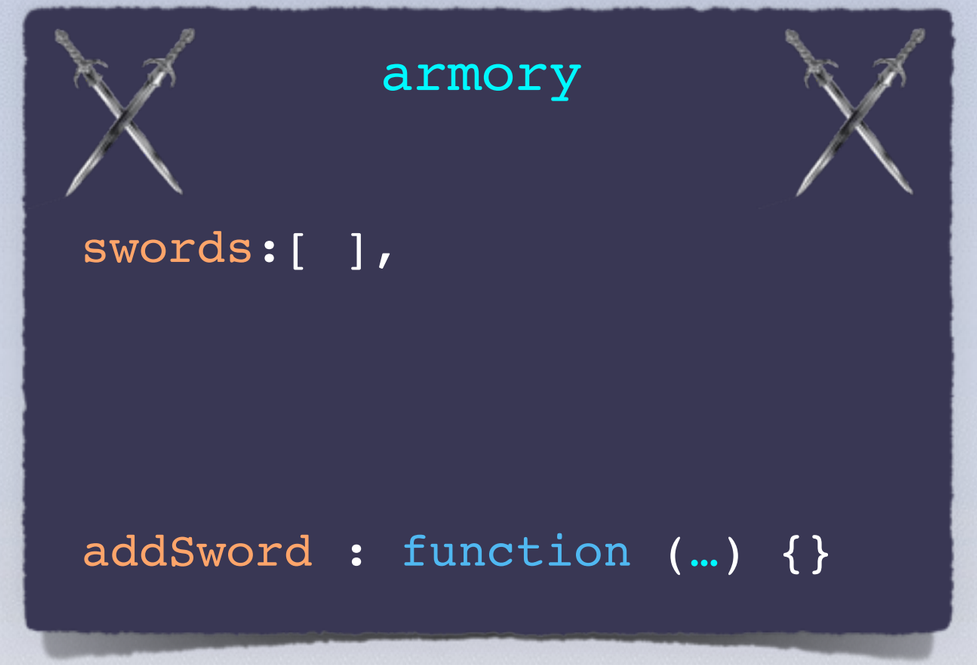
A CLOSER LOOK AT THAT “OR” ASSIGNMENT

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");
```



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");
```

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.

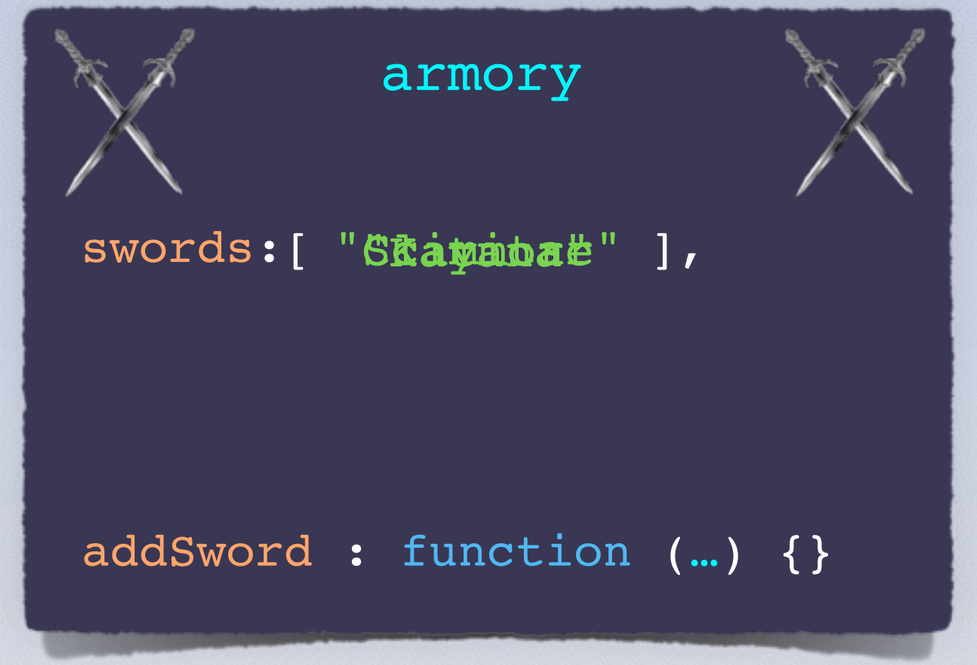
```
var armory = {   addSword: function (sword) {
                    this.swords = [ ] || this.swords;
                    this.swords.push(sword);
                 }
               };
```

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

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

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

```
["Scimitar"]
```



A CLOSER LOOK AT THAT “OR” ASSIGNMENT

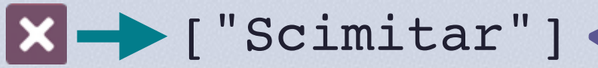
The OR operator takes the leftmost “truthy” value, and if none exists, the last “falsy” value.

```
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);
```



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);
```

→ undefined

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 I: The “OR” Operator

JAVASCRIPT
BEST PRACTICES

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

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

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

→ King

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

→ Arthur

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);
```

→ ""

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

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

→ ""

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

→ undefined

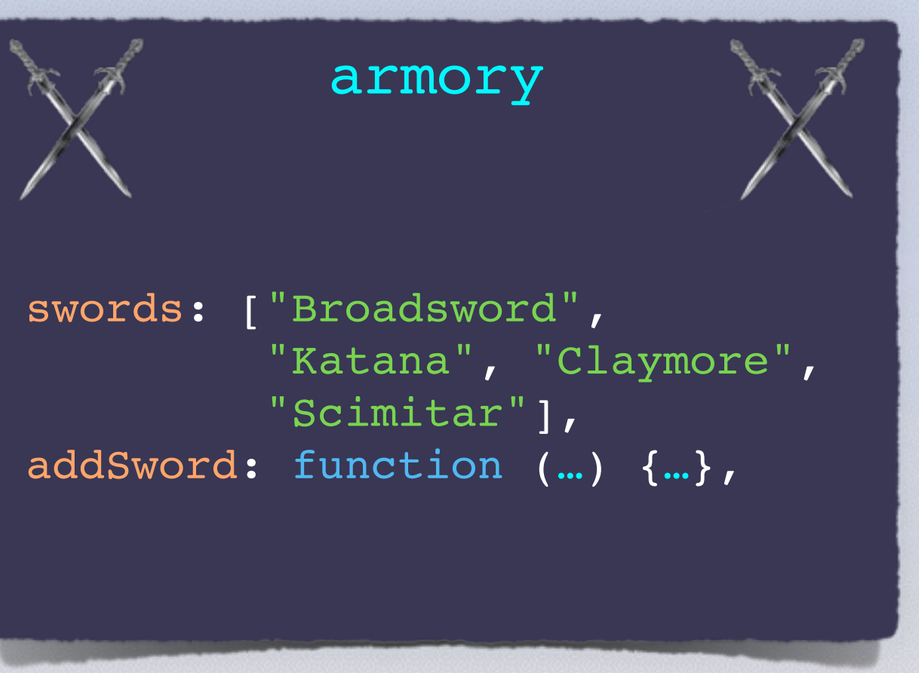
Again, compare that with OR, which still tries to find a “truthy” value somewhere before finally returning a “falsy” value as a last resort…

“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.

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 );
```

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.

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 );
```

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.

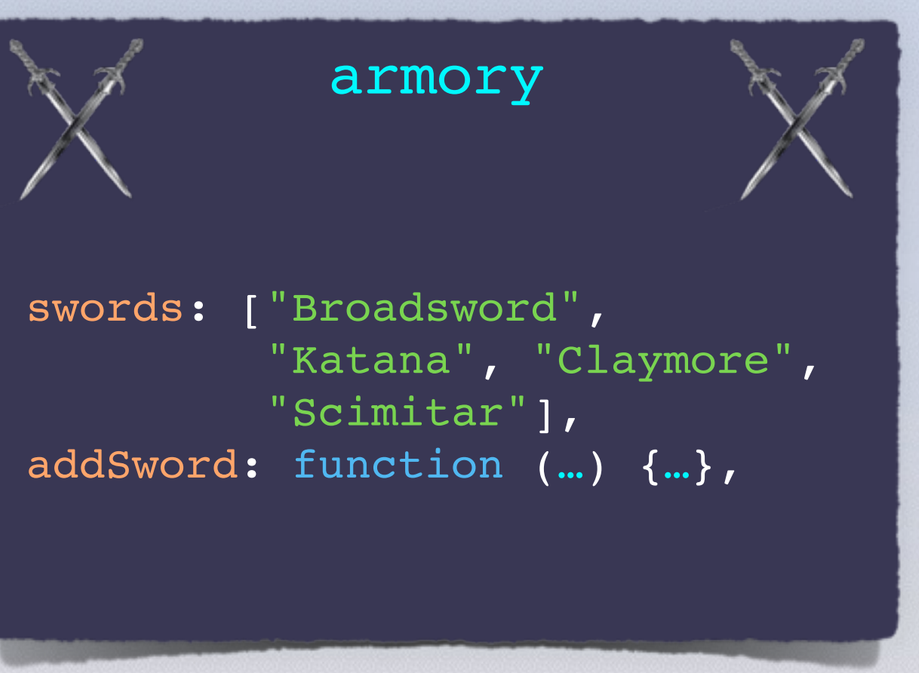
```
armory.retrieveSword = function (request){
    return (this.swords.indexOf(request) >= 0) ?
      this.swords.splice(this.swords.indexOf(request), 1)[0] :

};
```

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.

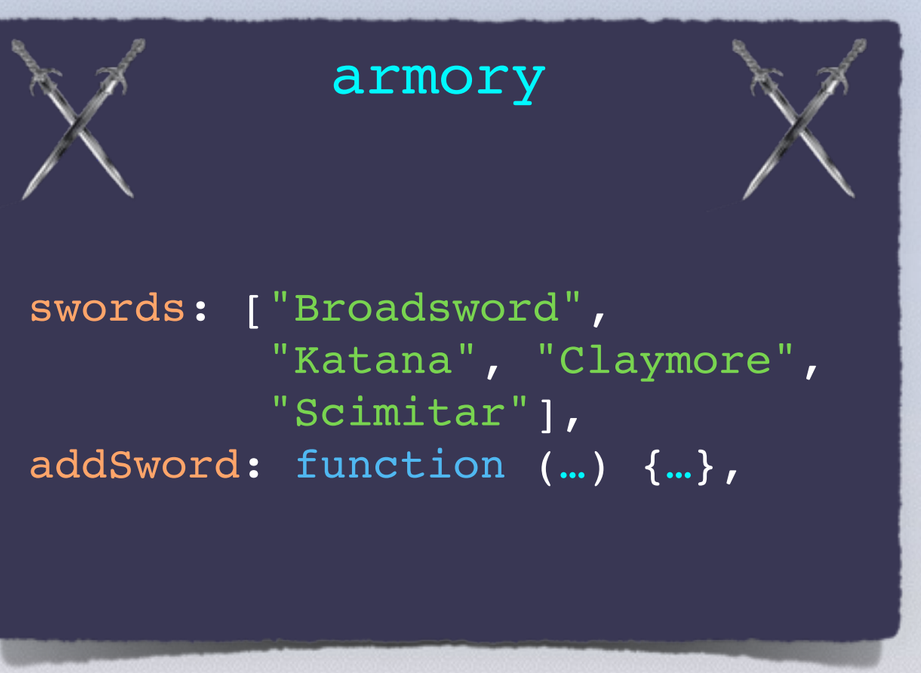


“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

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!") ;
};
```

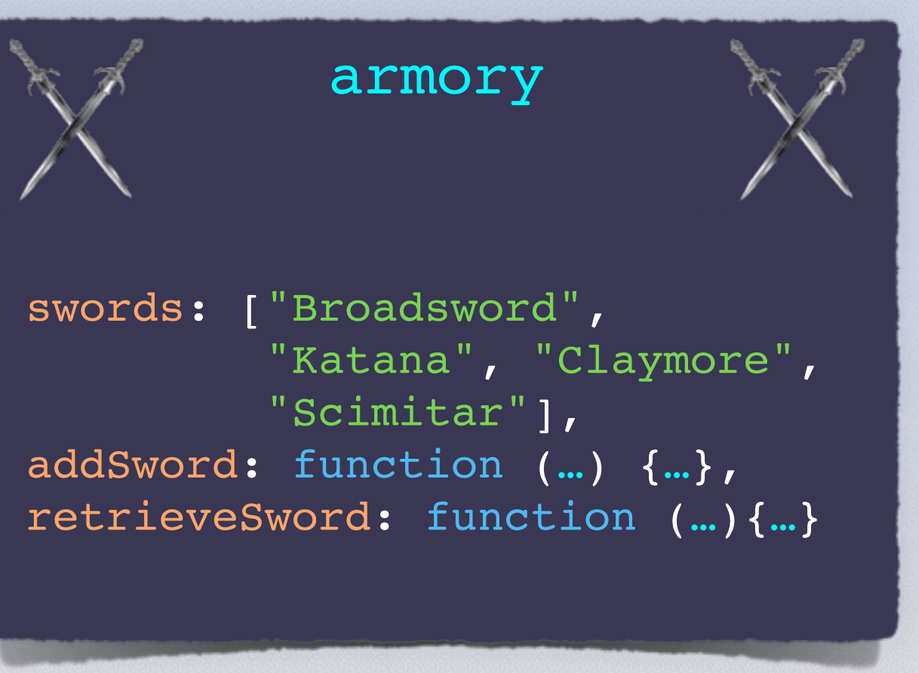
If we never initially found the sword, however, we’ll alert that this particular sharp thing ain’t here. This will be useful when we employ `&&`.



“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 (…){…}
```

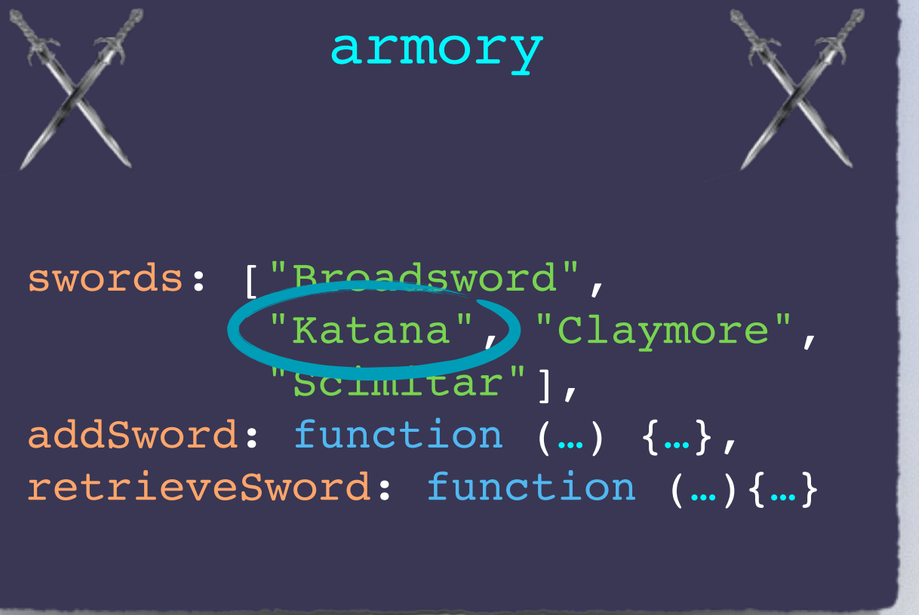
“AND” IS USEFUL IN CONTINGENT ASSIGNMENTS

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!") ;
};
```

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.

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



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

```
["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!") ;
};
```

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.

```
["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!") ;
};
```

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!") ;
};
```

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");
```

true

"Katana"

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

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");
```

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

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

→ false

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

armory

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

THE SWORD OF SYNTAX

Section 2

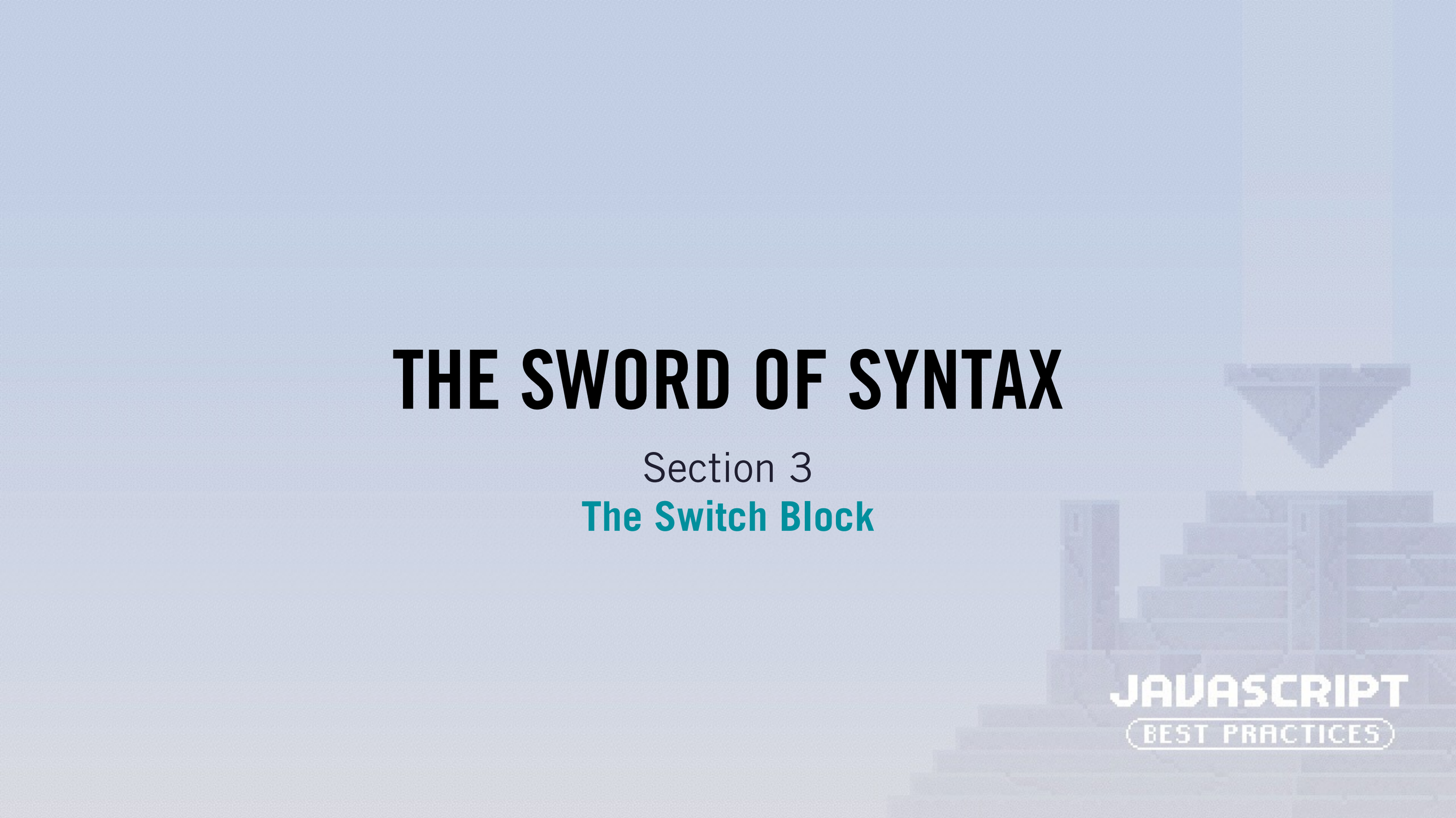
Logical Assignment II: The “AND” Operator

JAVASCRIPT BEST PRACTICES

THE SWORD OF SYNTAX

Section 3

The Switch Block



A CONDITIONAL FOR MULTIPLE POSSIBILITIES

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

`Broadsword`

`Claymore`

`Longsword`

`Mace`

`War Hammer`

`Battle Axe`

`Halberd`

`Morning Star`

A CONDITIONAL FOR MULTIPLE POSSIBILITIES

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


1


2


3


4


5

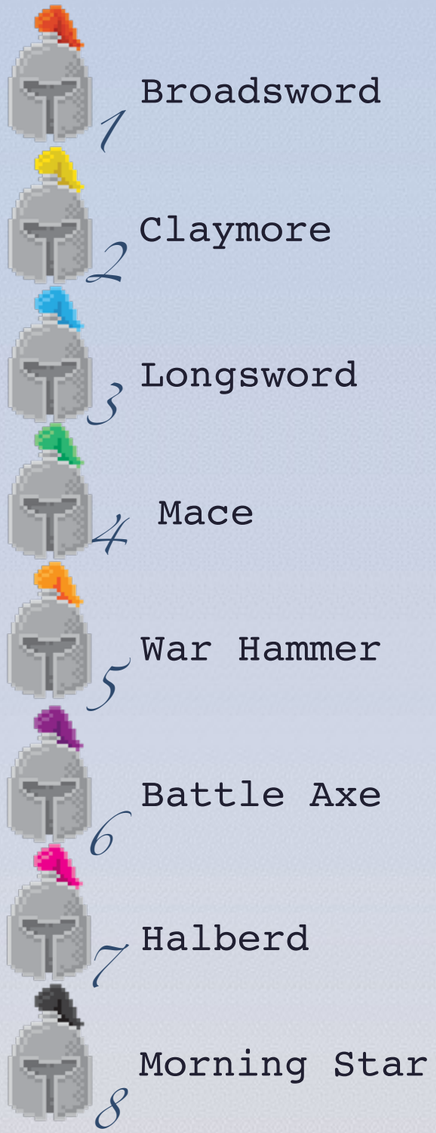

6


7


8

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.

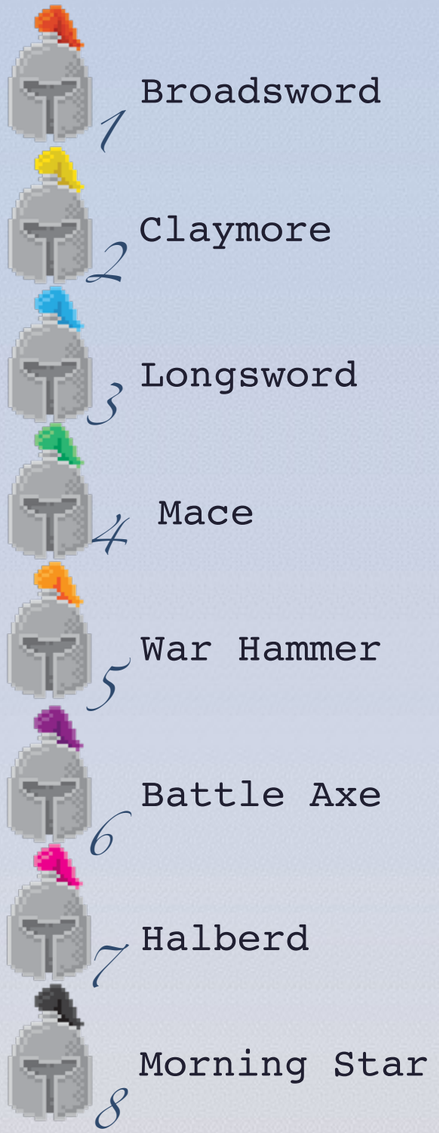
ENTER THE SWITCH KEYWORD

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.

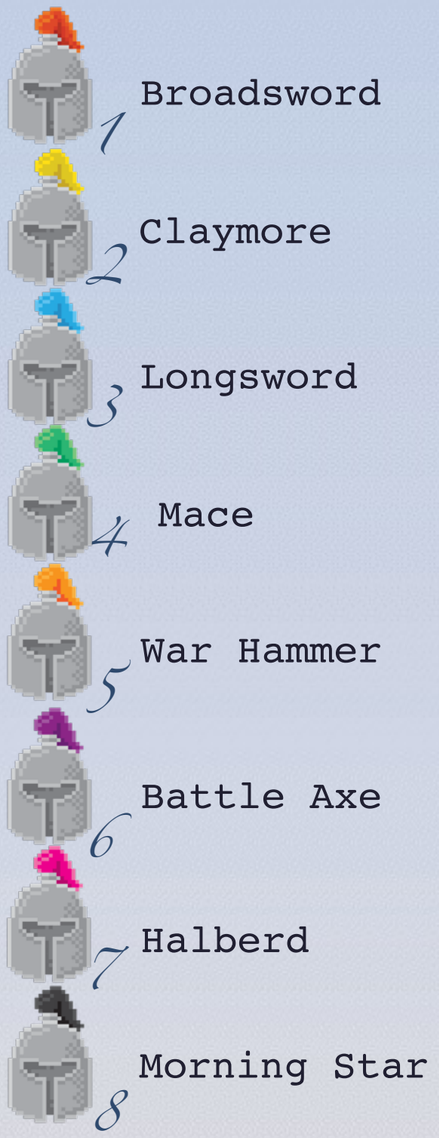



```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {



    }
}
```

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.

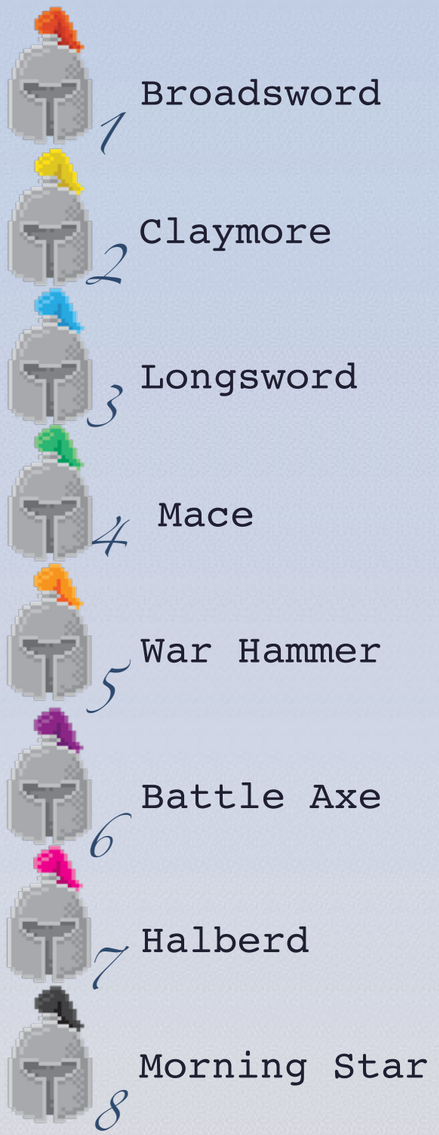



```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
      case 1:
        this.weapon = "Broadsword";



    }
}
```

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.

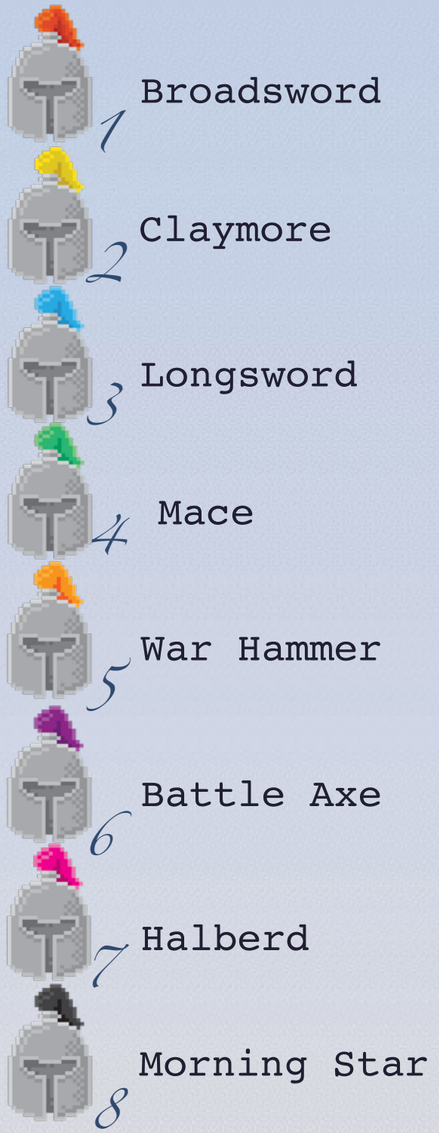



```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
      case 1:
        this.weapon = "Broadsword";

    }
}
```

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);
```

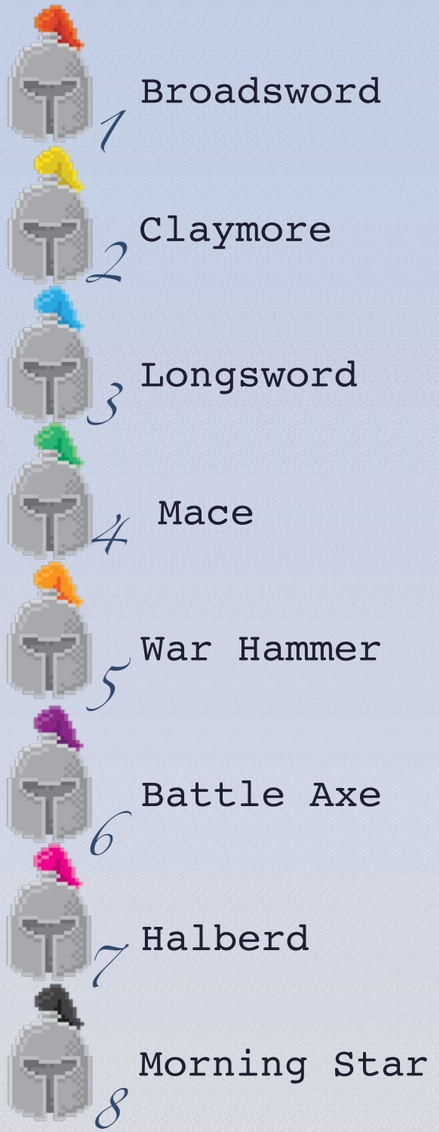
✗ → Morning Star

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";
    }
}
```

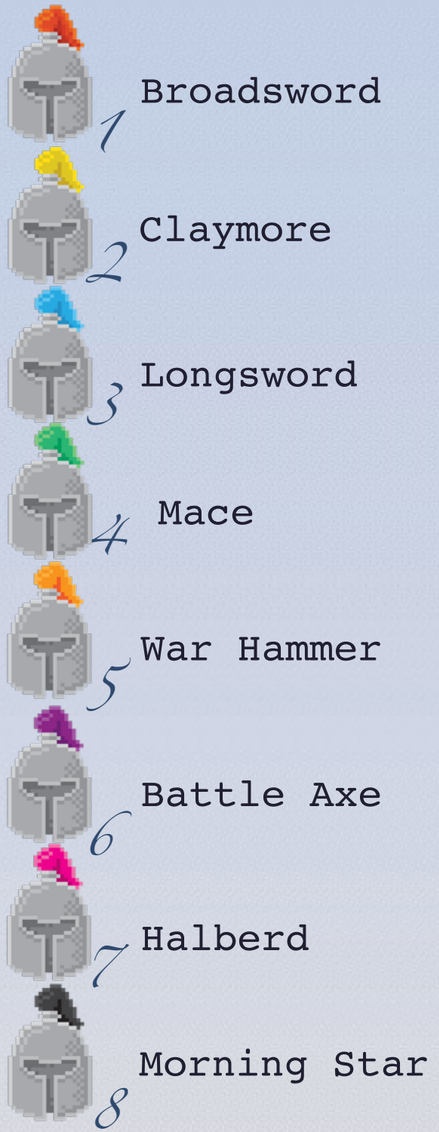
```
var soldier2 = new Knight("Richard", 4);
console.log(soldier2.weapon);
```

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”.



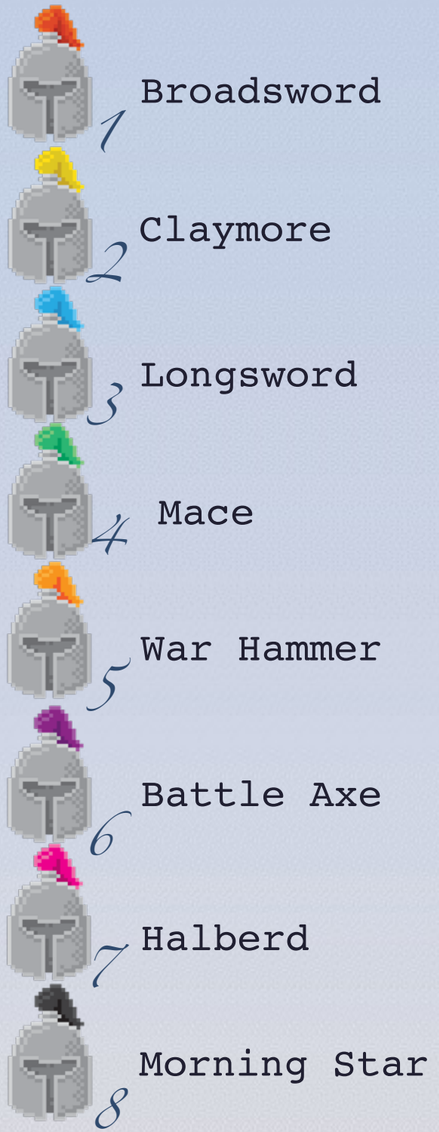

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment)
      case 1:
        this.weapon = "Broadsword";
      case 2:
        this.weapon = "Claymore";
      case 3:
        this.weapon = "Longsword";
      case 4:
        this.weapon = "Mace";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

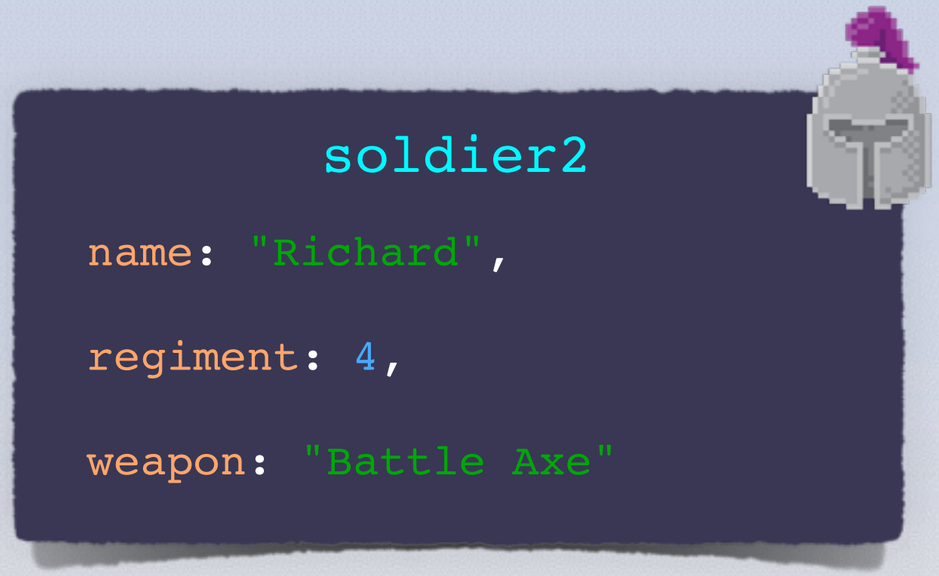
JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

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



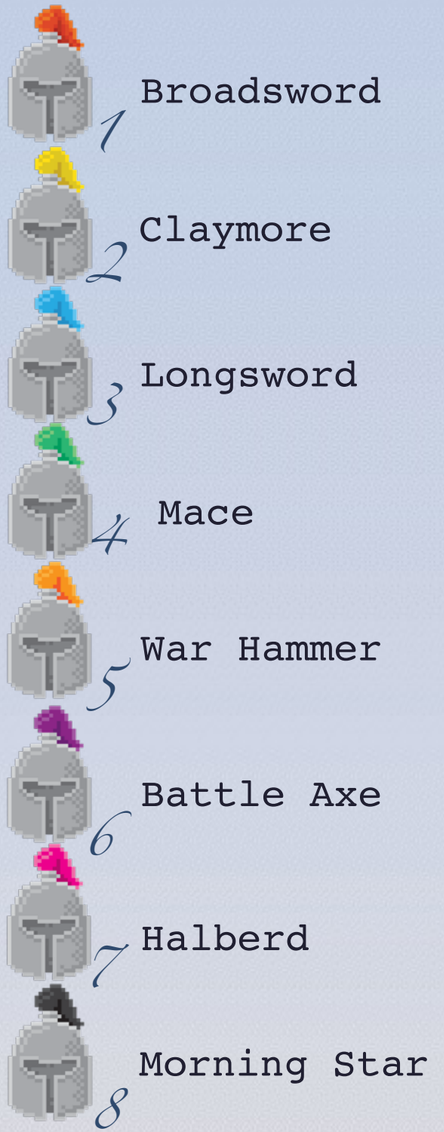

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment)
      case 1:
        this.weapon = "Broadsword";
      case 2:
        this.weapon = "Claymore";
      case 3:
        this.weapon = "Longsword";
      case 4:
        this.weapon = "Mace";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



JAVASCRIPT CASES ALLOW EXECUTION FALL-THROUGH

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



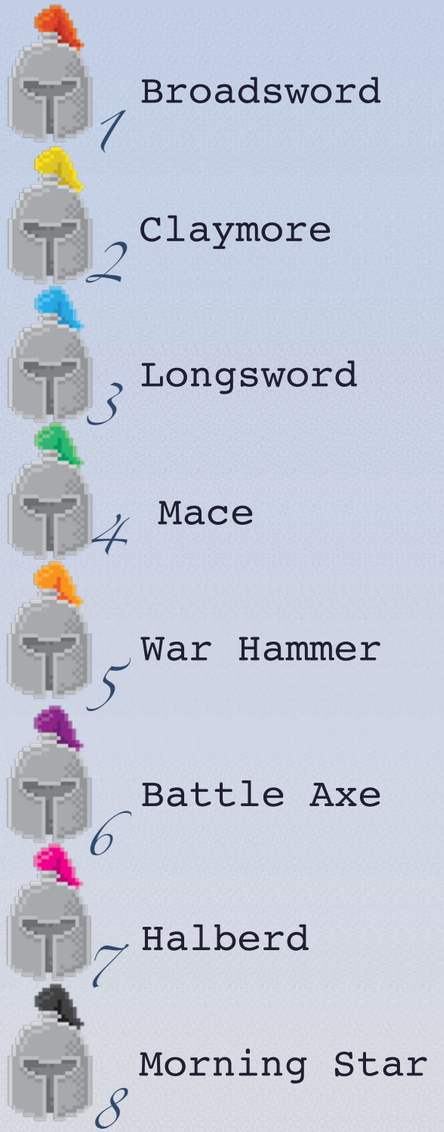

```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment)
        case 1:
          this.weapon = "Broadsword";
        case 2:
          this.weapon = "Claymore";
        case 3:
          this.weapon = "Longsword";
        case 4:
          this.weapon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

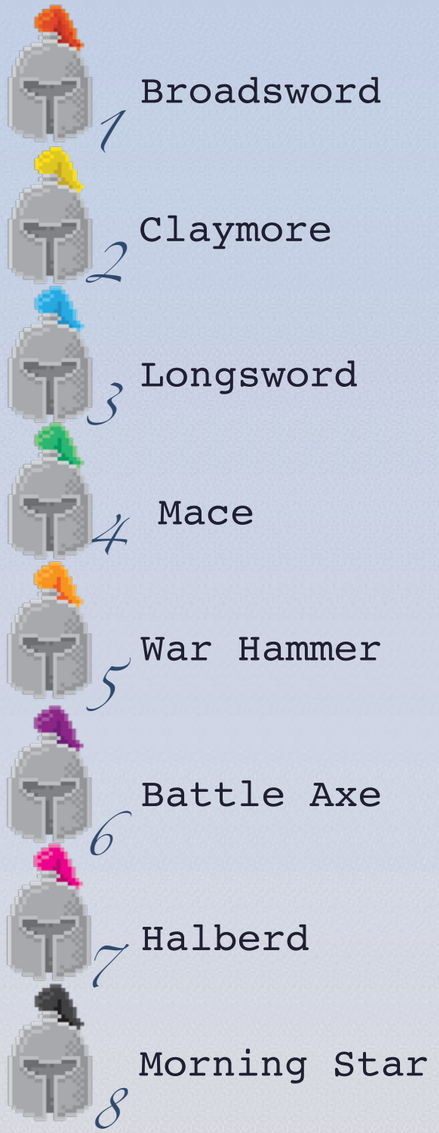
```
var soldier2 = new Knight("Richard", 4);
console.log(soldier2.weapon);
```

soldier2

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

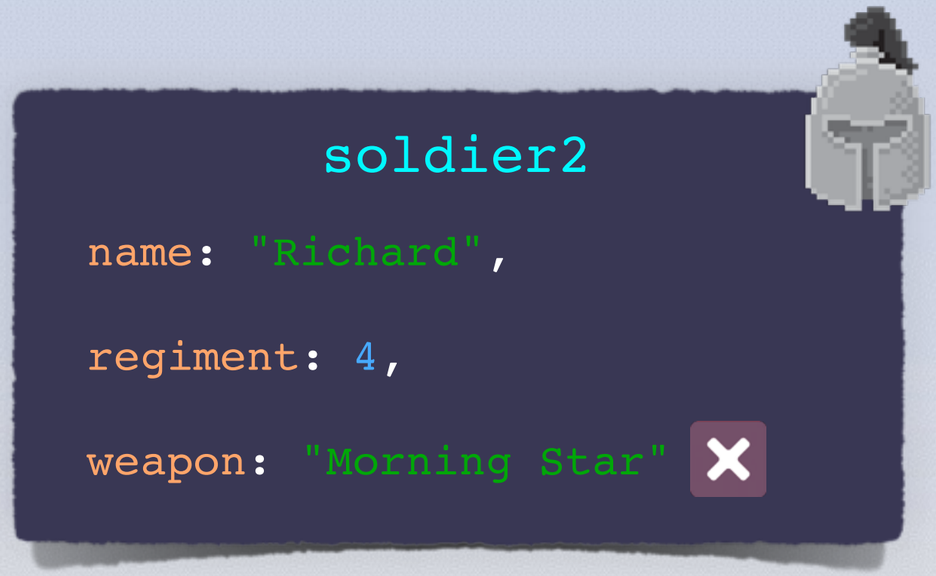
THE BREAK KEYWORD PROVIDES A QUICK EXIT

‘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);
```



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(soldier2.weapon);
```

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

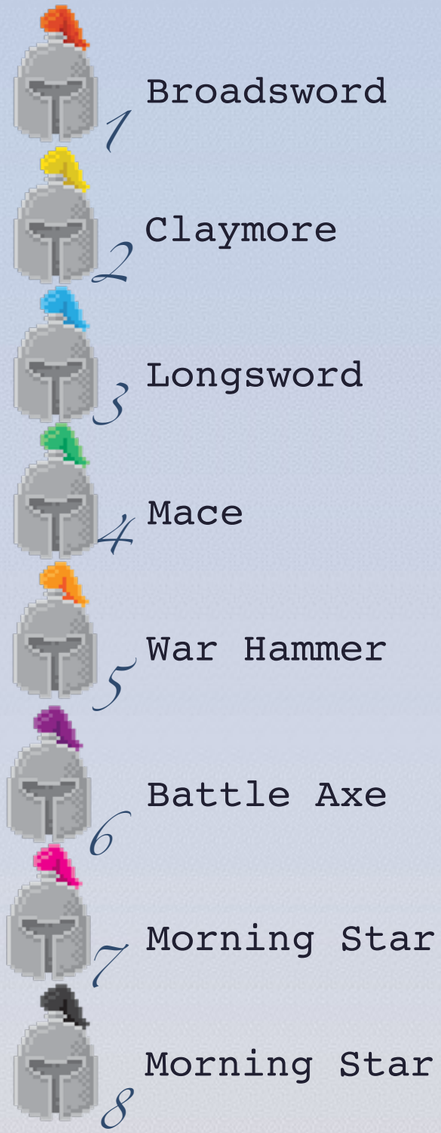
WHAT IF MULTIPLE CASES TAKE THE SAME ACTION?

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?

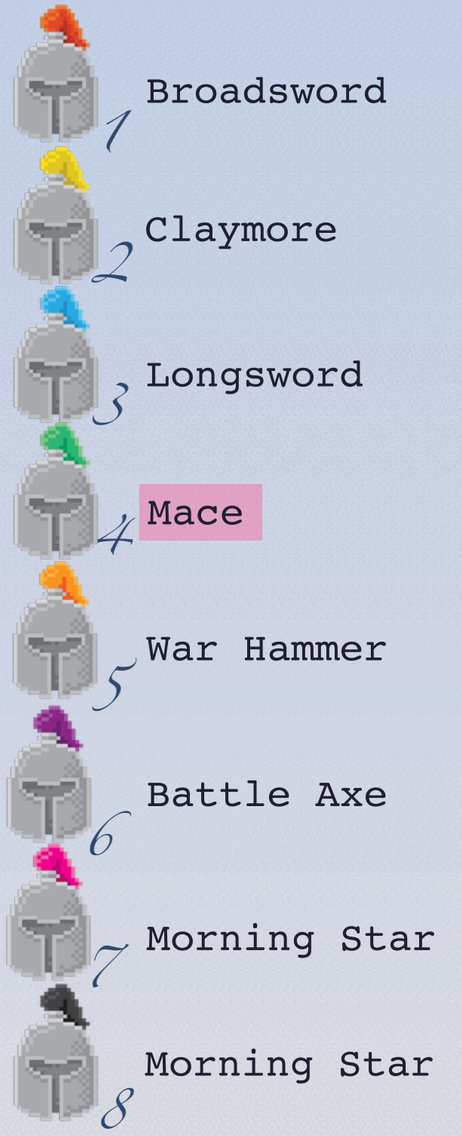
Fall-through will tighten up our code if we stack similar cases before their appropriate action.


```
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?

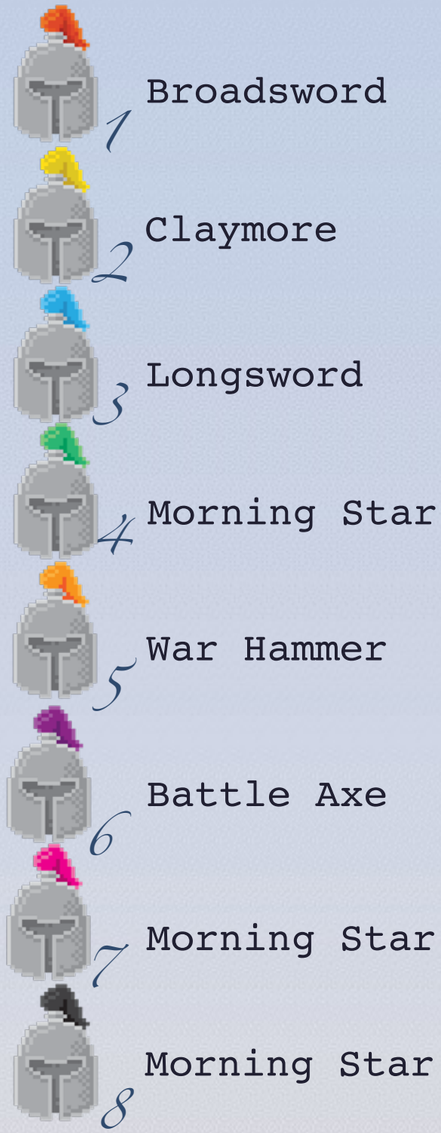
JavaScript doesn't care about case order, so we can also move around our numerical cases.




```
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?

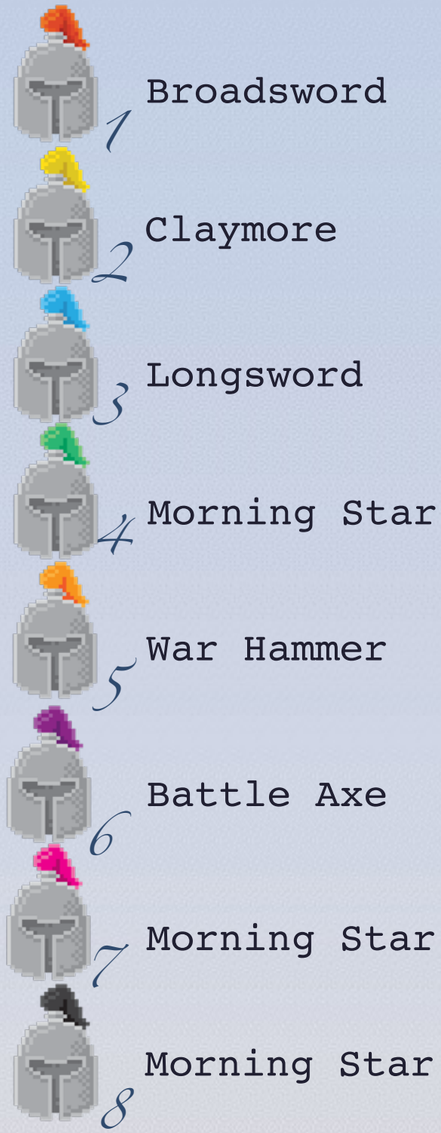
JavaScript doesn't care about case order, so we can also move around our numerical cases.




```
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?

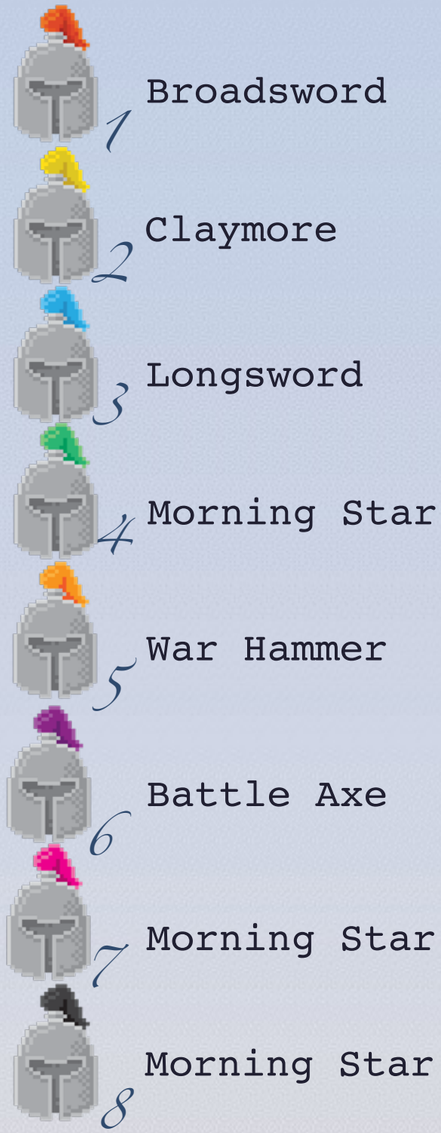
JavaScript doesn't care about case order, so we can also move around our numerical cases.




```
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?

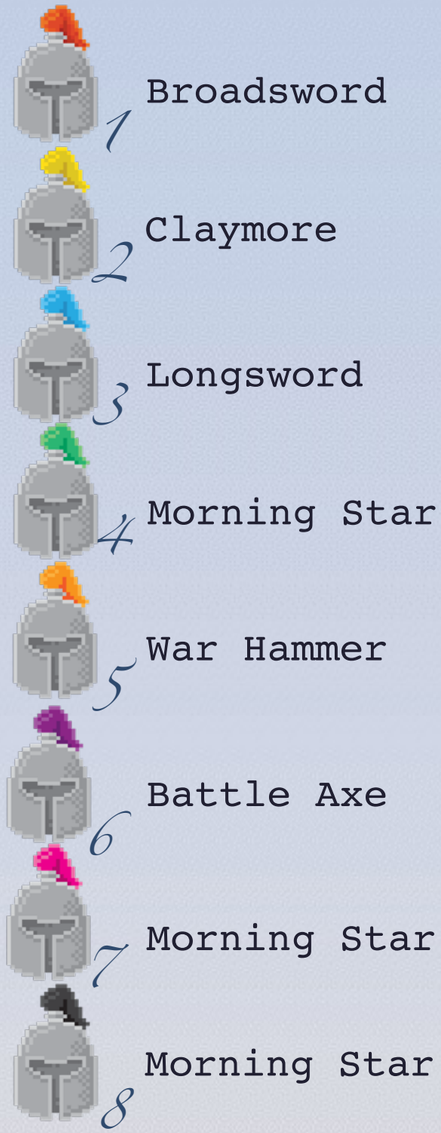
JavaScript doesn't care about case order, so we can also move around our numerical cases.




```
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.

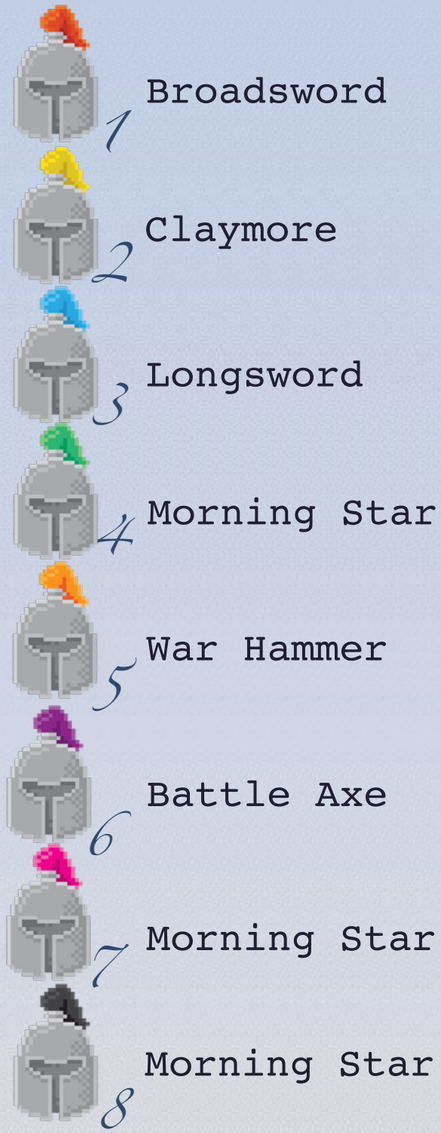



```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
      case 1:
        this.weapon = "Broadsword";
        break;
      case 2:
        this.weapon = "Claymore";
        break;
      case 3:
        this.weapon = "Longsword";
        break;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

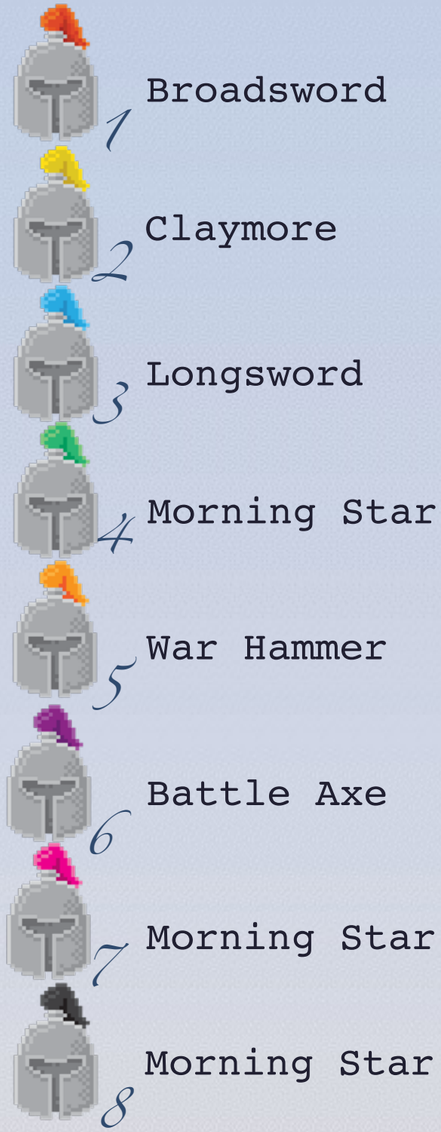
Fall-through!

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

✅ → Morning Star

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;
    }
}
```

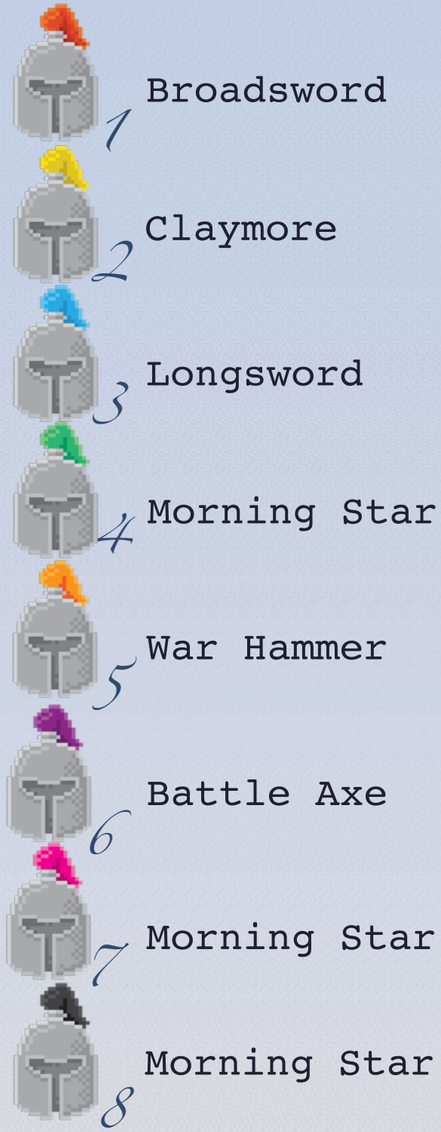
```
var king = new Knight("Arthur", "King");
console.log(king.weapon);
```

→ 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);
```

VERILY, A KNIGHT WITH NO WEAPON SUCKETH MUCH

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

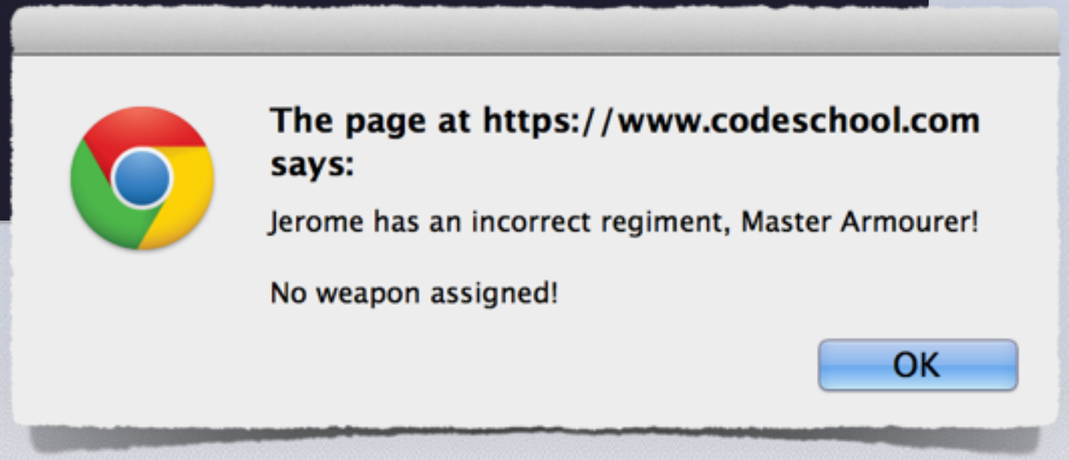
```
function Knight (name, regiment){
    this.name = name;
    this.regiment = regiment;
    switch (regiment) {
      case 1:
        this.weapon = "Broadsword";
        break;
      case 2:
        this.weapon = "Claymore";
        break;
      case 3:
        this.weapon = "Longsword";
        break;
      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;
      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 " +

    }
}
```

```
var soldier3 = new Knight("Jerome", 12);
```



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

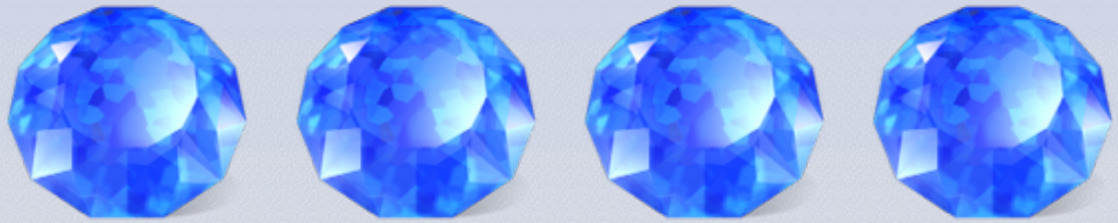
A carefully organized switch block can add LEAST common properties first and MOST common, last.

```
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 SWORD OF SYNTAX

Section 3

The Switch Block