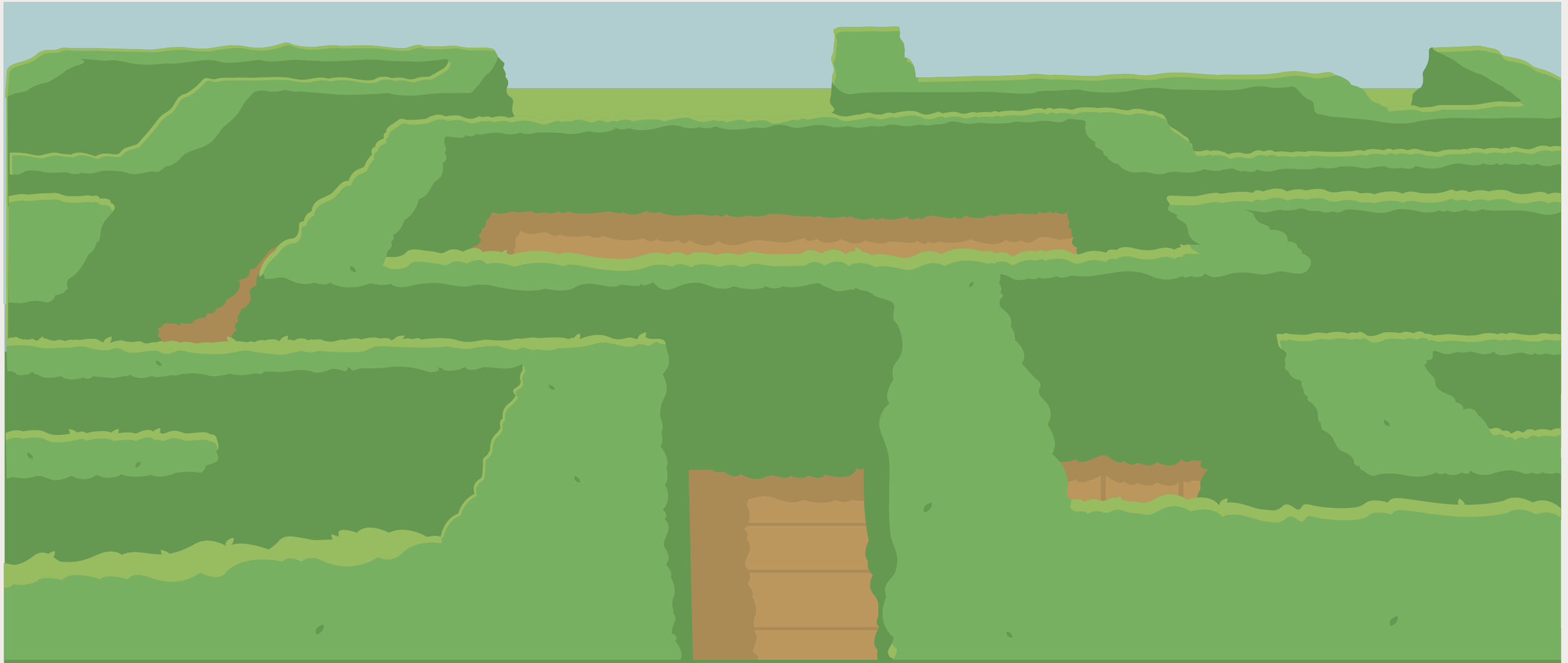


Get Lost In

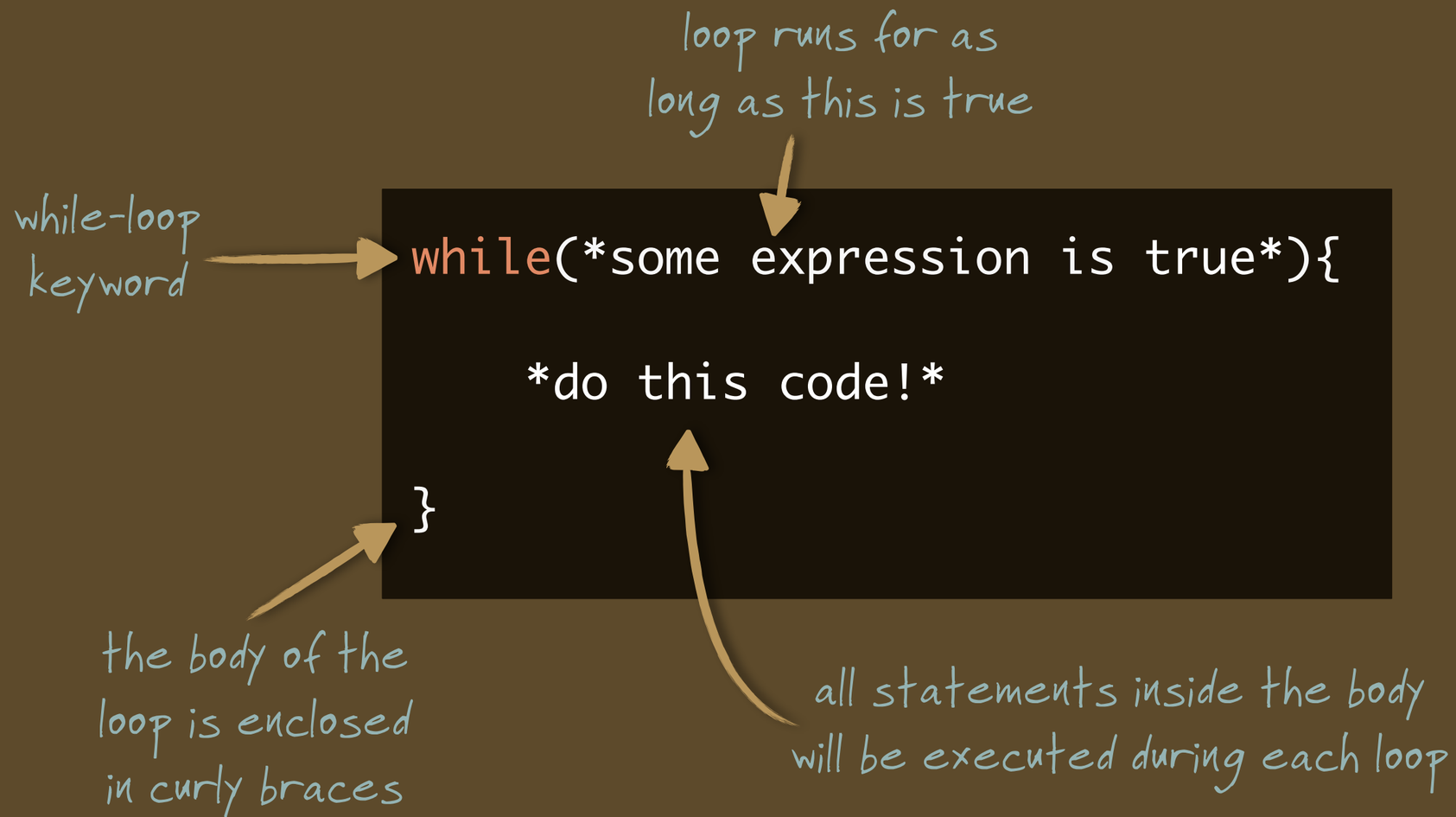
THE LABYRINTH OF LOOPS

LEVEL 1

THE LABYRINTH OF LOOPS

A BASIC WHILE-LOOP

The While-loop runs its code as long as its boolean expression is true



A BASIC WHILE-LOOP

This code runs forever...

```
while(true){

    *do this code!*

}
```

...and this code never runs!

```
while(false){

    *do this code!*

}
```


A BASIC WHILE-LOOP

Let's make a loop that prints the numbers 1-5 in ascending order.

```
var number = 1;
while (number <= 5) {

    console.log(number);
    number++;

}
```

Make sure to control your loop condition! If it never becomes false, you've got a dreaded infinite loop!

→ 1
→ 2
→ 3
→ 4
→ 5

UNDERSTANDING LOOPS

Loops allow us to have code executed repeatedly without extra typing

We want to execute the following code for every running train:

```
console.log("Train #" + <number> + " is running.");
```

Since we know variables can be changed, let's make one that will count our trains:

```
var trainNumber = 1;
```

Initially, we set the number to 1 for the first train.

Then, we need a way to make our train counter increase on every repetition:

```
trainNumber++;
```

Incrementing trainNumber will move forward to the next train.

So now, the following code needs to "loop" until all running trains have been listed:

Loop back

```
console.log("Train #" + trainNumber + " is running.");
trainNumber++;
```

We might also say that this code needs to run as long as: `trainNumber <= 8`

TRACING THROUGH THE LOOP WE NEED

Before the loop, start by initializing

```
var trainNumber = 1;
```

Loop only if

```
trainNumber <= 8
```

Perform this in each loop

```
console.log("Train #" + trainNumber + " is running.");
```

At the end of each loop

```
trainNumber++;
```

Advances the train number

```
var trainNumber = 1;
while( trainNumber <= 8){
  console.log("Train #" + trainNumber + " is running.");
  trainNumber++;
}
```

TRACING THROUGH THE LOOP WE NEED

```
var trainNumber = 1;
while( trainNumber <= 8){
  console.log("Train #" + trainNumber + " is running.");
  trainNumber++;
}
```

trainNumber	trainNumber <= 8 ?	Loop Output
1	TRUE	Train #1 is running.
2	TRUE	Train #2 is running.
3	TRUE	Train #3 is running.
4	TRUE	Train #4 is running.
5	TRUE	Train #5 is running.
6	TRUE	Train #6 is running.
7	TRUE	Train #7 is running.
8	TRUE	Train #8 is running.
9	FALSE	STOP!

BUILDING OUR WHILE-LOOP IN TRAINS.JS

Using variables instead of values to control our train-loop

By using variables, our loop is flexible. We could quickly change how many operational trains there are, and our loop would still print them all.

trains.js

```
var trainsOperational = 8;
var trainNumber = 1;
while (  trainNumber <= trainsOperational  ){

      console.log("Train #" + trainNumber + " is running.");
      trainNumber++;

}
```

THE FOR-LOOP

A different way of producing the same looping behavior

We typically use this spot for initialization.

This is usually the ++ or -- statement that controls the loop.

```
for (    *start with this*     ;    *loop if this expression is true*     ; *do this after each loop* ) {

    *in each loop, do this code!*

}
```

This is just like the condition that the While-loop checks before looping

THE FOR-LOOP

A different way of producing the same looping behavior

```
for ( var trainNumber = 1;  trainNumber <= trainsOperational;      trainNumber++      ) {

     console.log("Train #" + trainNumber + " is running.");

}
```


THE FOR-LOOP

Visualizing the flow of our for-loop

The loop counter is first initialized as part of the loop itself.

The loop checks an expression to see if another loop should execute.

Before another loop, the middle expression is checked again.

```
for ( var trainNumber = 1; trainNumber <= trainsOperational; trainNumber++ ) {


    console.log("Train #" + trainNumber + " is running.");


}
```

If the loop should run, the body code will execute.

After body code is executed, the final expression is executed before moving on. This statement usually affects whether the loop will run again.

FOR-LOOP IN ACTION

Printing numbers in descending order

Semicolons separate expressions.

This loop runs as long as number is still greater than zero.

```
for(var number = 5; number > 0; number--) {

    console.log(number);

}
```

Decreases number's value by one at the end of the loop.

Value of number	number > 0?	Loop Output
5	TRUE	5
4	TRUE	4
3	TRUE	3
2	TRUE	2
1	TRUE	1
0	FALSE	STOP!

IDENTIFYING THE NON-OPERATIONAL TRAINS

We’ve printed the running trains; now we turn to the stopped trains.

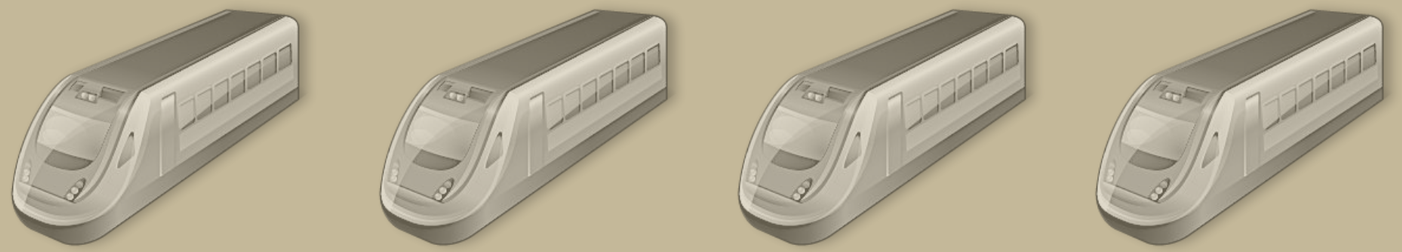
1 2 3 4 5 6

✓ ✓ ✓ ✓ ✓ ✓

7 8

✓ ✓

9 10 11 12



✗ ✗ ✗ ✗

WHICH TRAINS ARE NOT RUNNING?

Let's use the For-loop to list the non-operational trains

```
var trainsOperational = 8;
```

```
var totalTrains = 12;
```

Set stoppedTrain to the train number immediately after our last running train, i.e. #9

The loop should run only until we have reached the maximum # of trains

```
for(var stoppedTrain = trainsOperational + 1; stoppedTrain <= totalTrains; stoppedTrain++){

    console.log("Train #" + stoppedTrain + " is not operational.");

}
```

Wahoo, stopped trains only!

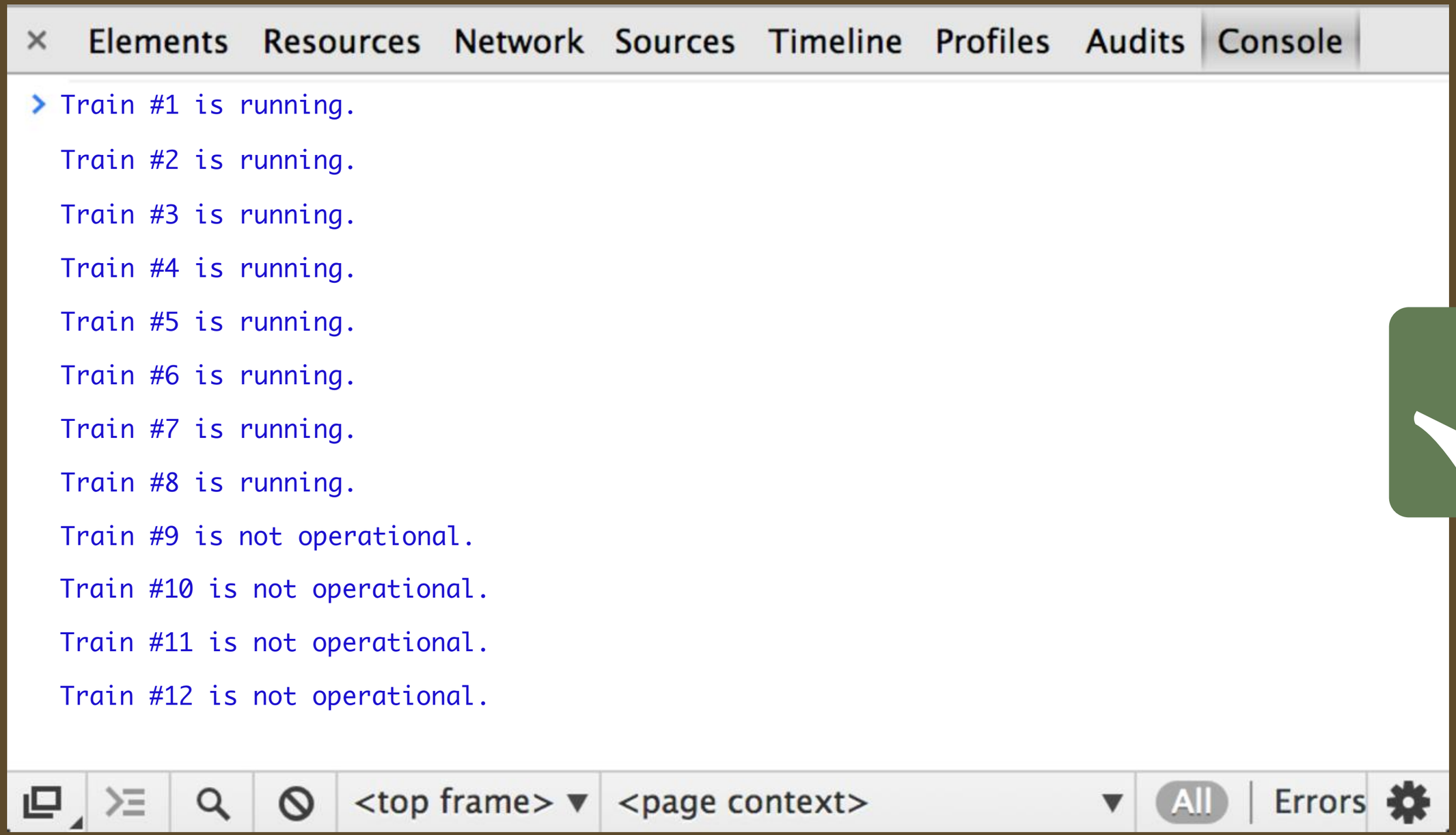
```
Train #9 is not operational.
Train #10 is not operational.
Train #11 is not operational.
Train #12 is not operational.
```

ADDING OUR FOR-LOOP TO TRAINS.JS

trains.js

```
var totalTrains = 12;
var trainsOperational = 8;


var trainNumber = 1;
while(trainNumber <= trainsOperational){
        console.log("Train #" + trainNumber + " is running.");
        trainNumber++;
}

for(var stoppedTrain = trainsOperational + 1; stoppedTrain <= totalTrains; stoppedTrain++){
        console.log("Train #" + stoppedTrain + " is not operational.");
}
```


RUNNING OUR CURRENT SOLUTION



Traverse the Ol'

CONDITIONAL CANYON

LEVEL 2

CONDITIONAL CANYON

THE CURRENT BUILD FOR OUR TRAIN STATUS SYSTEM

Our system currently prints both operational and non-operational trains

1  ✓

2  ✓

3  ✓

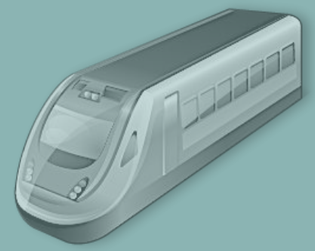
4  ✓

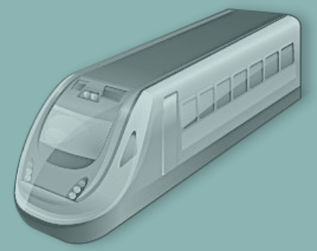
5  ✓

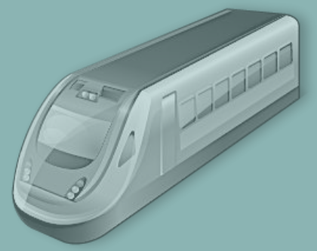
6  ✓

7  ✓

8  ✓

9  ✗

10  ✗

11  ✗

12  ✗

OUR CURRENT SYSTEM USES TWO LOOPS...

trains.js

```
var totalTrains = 12;
var trainsOperational = 8;

var trainNumber = 1;
while(trainNumber <= trainsOperational){
        console.log("Train #" + trainNumber + " is running.");
        trainNumber++;
}

for(var stoppedTrain = trainsOperational + 1; stoppedTrain <= totalTrains; stoppedTrain++){
        console.log("Train #" + stoppedTrain + " is not operational.");
}
```

We want to merge these loops into one loop that DECIDES which trains are running and which aren't.

LETS USE ONE LOOP INSTEAD OF TWO

So how do we run different lines of code based on specific conditions?

trains.js

```
...
for (var trainNumber = 1; trainNumber <= totalTrains; trainNumber++){

    *if the train is currently running, we want:*
        console.log("Train #" + trainNumber + " is running.");

    *otherwise, if it's not, we want:*
        console.log("Train #" + trainNumber + " is not operational.");
}
```

Notice now that we're now using only one variable to identify the train's number.

IF, AND HER BUDDY, ELSE

If and Else allow us to execute certain code based on specific conditions

If the conditional evaluates to true, the code block is executed.

```
if (*some condition is true*) {

 *do this code!*

} else {

  *OTHERWISE, do this code instead!*

}
```

Else follows up with code to execute ONLY when the If conditional is not satisfied. It is ignored otherwise.

IF, AND HER BUDDY, ELSE

A basic example of conditional execution

```
var value1 = 4;
var value2 = 9;

if ( value1 < value2 ) {

    console.log(value1 + " is less than " + value2);

} else {

    console.log(value1 + " is greater than or equal to " + value2);

}
```

We aren't sure whether it's strictly greater than, only that it is not less than.

→ 4 is less than 9

IF, AND HER BUDDY, ELSE

A basic example of conditional execution

```
var value1 = 12;
var value2 = 9;

if ( value1 < value2 ) {

    console.log(value1 + " is less than " + value2);

} else {

    console.log(value1 + " is greater than or equal to " + value2);

}
```

Now, this conditional will evaluate to false, and so the 'else' block will trigger.

```
→ 12 is greater than or equal to 9
```

Trust us on this.

CAN IF AND ELSE MAKE OUR TRAINS.JS BETTER?

Using conditionals for efficiency

trains.js

```
var totalTrains = 12;
var trainsOperational = 8;


var trainNumber = 1;
while (trainNumber <= trainsOperational){
        console.log("Train #" + trainNumber + " is running.");
        trainNumber++;
}
for (var stoppedTrain = trainsOperational + 1; stoppedTrain <= totalTrains; stoppedTrain++) {
        console.log("Train #" + stoppedTrain + " is not operational.");
}
```


BUILDING OUR NEW SYSTEM STATUS LOOP

Looping with If and Else controls

trains.js

```
...
for (var trainNumber = 1; trainNumber <= totalTrains; trainNumber++){

      *if the train is currently running, we want:*
            console.log("Train #" + trainNumber + " is running.");

      *otherwise, if it's not, we want:*
            console.log("Train #" + trainNumber + " is not operational.");
}
```


BUILDING OUR NEW SYSTEM STATUS LOOP

Looping with If and Else controls

trains.js

```
...
for (var trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {

    if (trainNumber <= operationalTrains) {
        console.log("Train #" + trainNumber + " is running.");
    } else {
        console.log("Train #" + trainNumber + " is not operational.");
    }
}
```

As soon as trainNumber is no longer within the amount of operational trains, the If block is skipped and the Else block begins printing in each new loop.

RUNNING OUR NEW SINGLE LOOP!

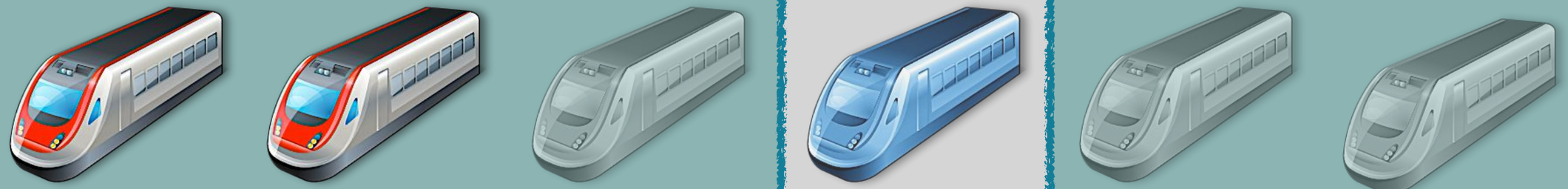
trainNumber	LOOP: trainNumber<=12?	Is trainNumber<=8?	OUTPUT
 1	TRUE	YES -> IF	Train #1 is running.
 2	TRUE	YES -> IF	Train #2 is running.
 3	TRUE	YES -> IF	Train #3 is running.
 4	TRUE	YES -> IF	Train #4 is running.
 5	TRUE	YES -> IF	Train #5 is running.
 6	TRUE	YES -> IF	Train #6 is running.
 7	TRUE	YES -> IF	Train #7 is running.
 8	TRUE	YES -> IF	Train #8 is running.
 9	TRUE	NO -> ELSE	Train #9 is not operational.
 10	TRUE	NO -> ELSE	Train #10 is not operational.
 11	TRUE	NO -> ELSE	Train #11 is not operational.
 12	TRUE	NO -> ELSE	Train #12 is not operational.
 13	FALSE	STOP THE LOOP!	

ADDING A SPECIAL TRAIN THAT STARTS LATER

Let’s add a train that isn’t operational yet, but starts at noon.

1 2 3 4 5 6

7 8 9 10 11 12



THE ELSE-IF SYNTAX

When two conditions just isn't enough!

```
if (*some condition is true*) {

  *do this code!*

} else if (*some OTHER condition is true*) {

   *do something for this condition!*

} else {

   *IN ALL OTHER CASES, do this code instead!*

}
```

Remember that as soon as a condition is met in any block, the rest will be skipped entirely!

CHECKING MULTIPLE CONDITIONS

“Else If” can be used when many specific scenarios need attention

```
if (trainNumber <= operationalTrains) {

     console.log("Train #" + trainNumber + " is running.");

} *otherwise, first check if the train is the express train*{

     console.log("Train #10 begins running at noon.");

} else {

     console.log("Train #" + trainNumber + " is not operational.");

}
```

CHECKING MULTIPLE CONDITIONS

“Else If” can be used when many specific scenarios need attention

```
if (trainNumber <= operationalTrains) {

      console.log("Train #" + trainNumber + " is running.");

} else if (trainNumber == 10) {

      console.log("Train #10 begins running at noon.");

} else {

      console.log("Train #" + trainNumber + " is not operational.");

}
```

This condition is checked ONLY when a train is NOT an operational train. Thus, train 10 only starts at noon if it is not ALREADY an operational train.

Triggers only when a train is neither regular NOR express

UPDATING OUR SYSTEM STATUS LOOP

Now we can print based on multiple conditions!

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {

  if (trainNumber <= operationalTrains) {
          console.log("Train #" + trainNumber + " is running.");
  } else if (trainNumber == 10) {
          console.log("Train #10 begins running at noon.");
  } else {
          console.log("Train #" + trainNumber + " is not operational.");
  }

}
```


SO WHAT DOES THIS GET US?

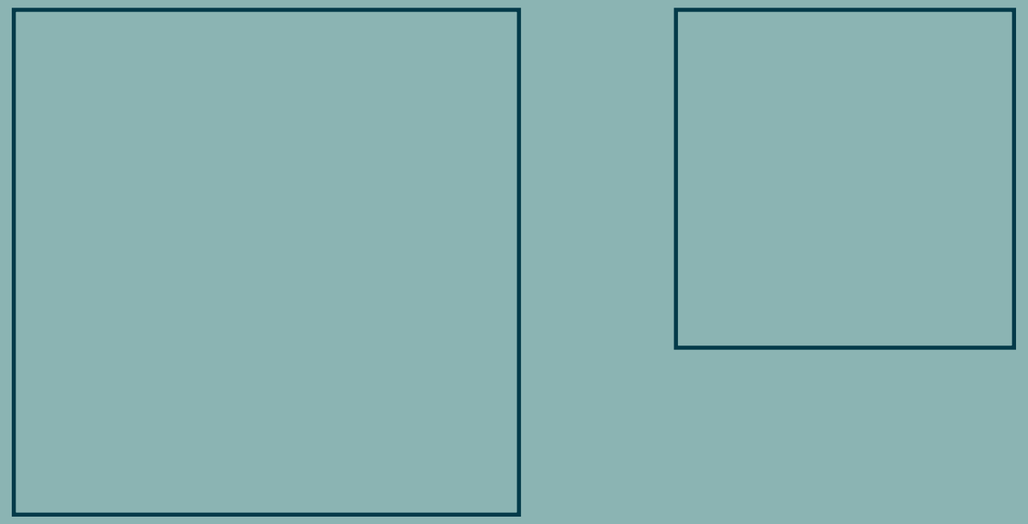
trainNumber	LOOP: trainNumber<=12?	Is trainNumber<=8?	Is trainNumber ==10?	OUTPUT
🚄 1	TRUE	YES -> IF	IGNORE	Train #1 is running.
🚄 2	TRUE	YES -> IF	IGNORE	Train #2 is running.
🚄 3	TRUE	YES -> IF	IGNORE	Train #3 is running.
🚄 4	TRUE	YES -> IF	IGNORE	Train #4 is running.
🚄 5	TRUE	YES -> IF	IGNORE	Train #5 is running.
🚄 6	TRUE	YES -> IF	IGNORE	Train #6 is running.
🚄 7	TRUE	YES -> IF	IGNORE	Train #7 is running.
🚄 8	TRUE	YES -> IF	IGNORE	Train #8 is running.
🚈 9	TRUE	NO	NO -> ELSE	Train #9 is not operational.
🚅 10	TRUE	NO	YES -> ELSE-IF	Train #10 begins running at noon.
🚈 11	TRUE	NO	NO -> ELSE	Train #11 is not operational.
🚈 12	TRUE	NO	NO -> ELSE	Train #12 is not operational.
❌ 13	FALSE	STOP THE LOOP!		

NESTED CONDITIONALS

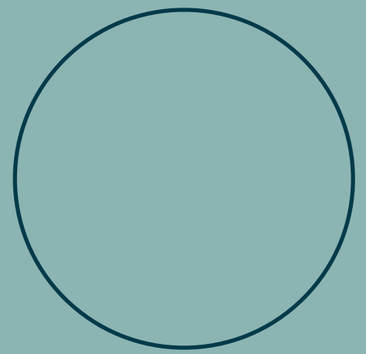
Splitting results for a single condition

Let’s say we had some shapes.

We have two sizes for squares...



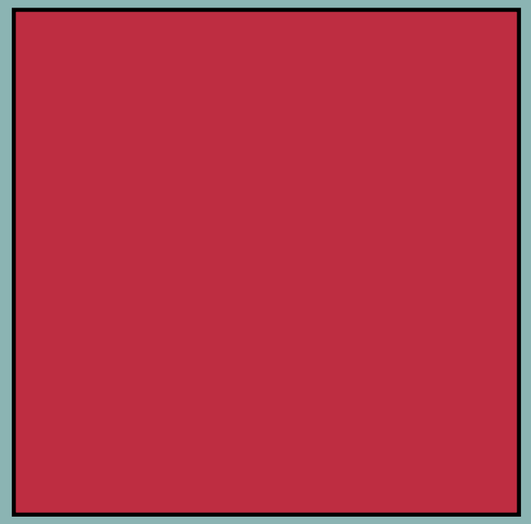
...but only one size for circles.

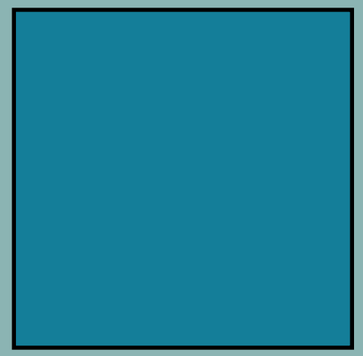


NESTED CONDITIONALS

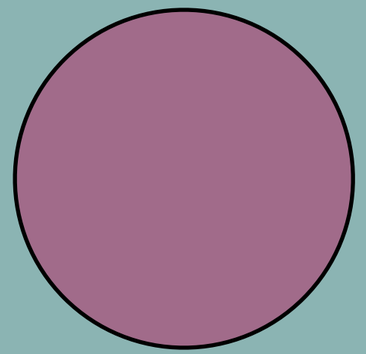
Splitting results for a single condition

We want to color big squares red,
and small squares blue...





...while all circles are purple.



NESTED CONDITIONALS

Splitting results for a single condition

```
if ( *it's a square* ) {

      if (*it's big*) {

         *make it red!*

      } else {

         *it must be a small square, so make it blue!*

      }

} else {

      *since its not a square, it must be a circle, so make it purple!*

}
```

This Else ONLY reacts to a failure of the most recently encountered If statement

This Else ONLY triggers if the very first If does not.

NESTED CONDITIONALS

Splitting results for a single condition

```
if (*there are ANY running trains*) {

      if (*the amount of running trains equals the amount of total trains*) {
         *print out to passengers that all trains are running!*
      } else {
         *just execute our existing loop code covering the status of trains*
      }

} else {
     *there must be no running trains, so print that out!*
}
```


NESTED CONDITIONALS

Splitting results for a single condition

```
if (trainsOperational > 0) {
    if (trainsOperational == totalTrains) {
        console.log("All trains are running at the JavaScript Express!");
    } else {
        for (var trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
            ...
        }
    }
} else {
        console.log("No trains are operational today. Bummer!");
}
```

Our already existing status conditionals from trains.js go here

UPDATING TRAINS.JS WITH NEW CONDITIONS

Now passengers will know if all trains are running, or if none are.

trains.js

Every possible message is still controlled by these two values!

```
var totalTrains = 12;
var trainsOperational = 8;

if ( trainsOperational > 0 ) {

    if (trainsOperational == totalTrains) {

        console.log("All trains are running at the JavaScript Express!");

    } else {

        for (var trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {

            ...

        }

    }

} else {

    console.log("No trains are operational today. Bummer!");

}
```

ALL TRAINS, OR NONE

Let's look at how we'd get these situations...

```
var totalTrains = 12;
var trainsOperational = 12;
```

```
trainsOperational > 0
```

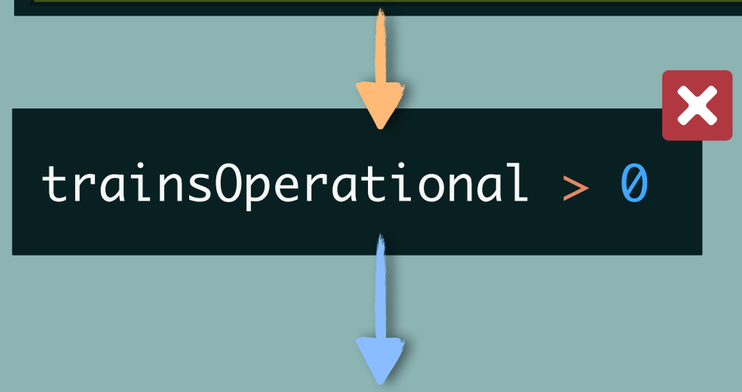
```
trainsOperational == totalTrains
```

```
All trains are running at the JavaScript Express!
```

ALL TRAINS, OR NONE

Let’s look at how we’d get these situations…

```
var totalTrains = 12;
var trainsOperational = 0;
```



```
trainsOperational > 0
```

```
No trains are operational today. Bummer!
```

ADDING TO OUR LIST OF SPECIAL TRAINS

Another train that will start running at noon on its non-operational days

1  ✓

2  ✓

3  ✓

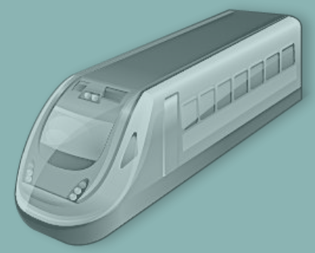
4  ✓

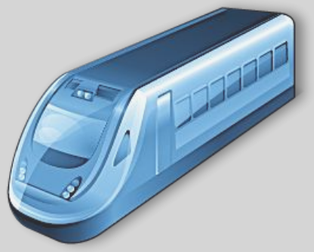
5  ✓

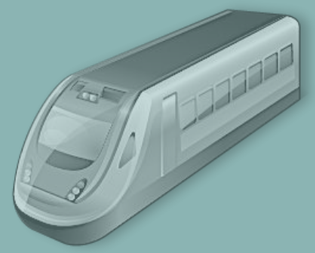
6  ✓

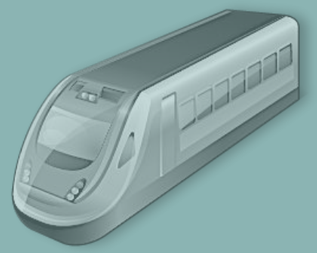
7  ✓

8  ✓

9  ✗

10  ✓

11  ✗

12  ✗

ADDING TO OUR LIST OF SPECIAL TRAINS

Another train that will start running at noon on its non-operational days

1  ✓

2  ✓

3  ✓

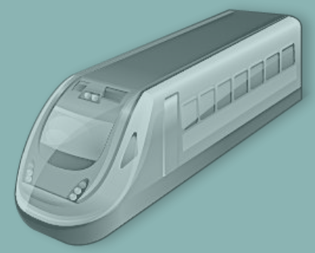
4  ✓

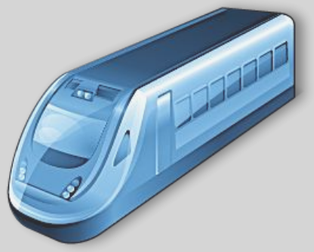
5  ✓

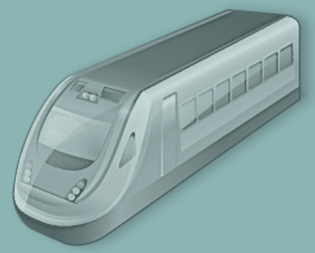
6  ✓

7  ✓

8  ✓

9  ✗

10  ✓

11  ✗

ADDING TO OUR LIST OF SPECIAL TRAINS

Another train that will start running at noon on its non-operational days

1  ✓

2  ✓

3  ✓

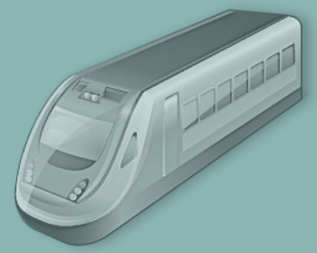
4  ✓

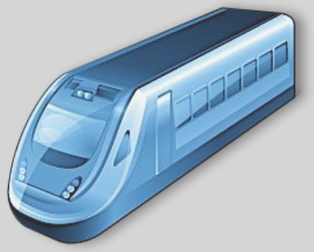
5  ✓

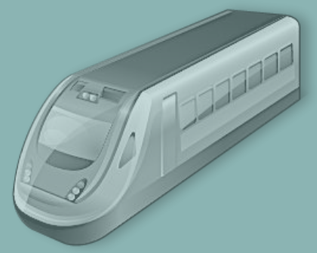
6  ✓

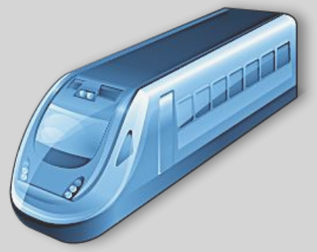
7  ✓

8  ✓

9  ✗

10  ✓

11  ✗

12  ✓

ADDING A SECOND CONDITION

Printing based on multiple conditions

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains) {
            console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10   *we want something else here*) {
            console.log("Train #10 begins running at noon.");
     } else {
            console.log("Train #" + trainNumber + " is not operational.");
     }
}
```


COMPLEX CONDITIONALS

&& **Binary ‘And’ returns true if BOTH values are true**

|| **Binary ‘Or’ returns true if EITHER value is true**

```
> true && false
```
→ false

```
> true && true
```
→ true

```
> false && false
```
→ false

```
> false || true
```
→ true

```
> false || false
```
→ false

```
> true || true
```
→ true

COMPLEX CONDITIONALS

&& **Binary ‘And’ returns true if BOTH values are true**

|| **Binary ‘Or’ returns true if EITHER value is true**

```
> ( 11 >= 11 ) && ( -7 < 6 )
```

true && true

→ true

```
> ( 2 >= 0 ) && ( 9 < 4 )
```

true && false

→ false

```
> ( 5 < 7 ) || ( 8 > 10 )
```

true || false

→ true

```
> ( 3 > 8 ) || ( 7 < 3 )
```

false || false

→ false

ADDING A SECOND CONDITION

Printing based on multiple conditions

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains) {
             console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10    *we want something else here*) {
             console.log("Train #10 begins running at noon.");
     } else {
             console.log("Train #" + trainNumber + " is not operational.");
     }

}
```


ADDING A SECOND CONDITION

Printing based on multiple conditions

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains) {
             console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10 || trainNumber == 12 ) {
             console.log("Train #" + trainNumber + " will begin running at noon.");
     } else {
             console.log("Train #" + trainNumber + " is not operational.");
     }

}
```

Now we use the trainNumber variable instead of hard-coding the values.

OUR UPDATED STATUS LOOP

All trains that begin running at noon are now printed!

trainNumber	LOOP: trainNumber<=12?	Is trainNumber<=8?	Is trainNumber ==10?	OUTPUT
1	TRUE	YES -> IF	IGNORE	Train #1 is running.
2	TRUE	YES -> IF	IGNORE	Train #2 is running.
3	TRUE	YES -> IF	IGNORE	Train #3 is running.
4	TRUE	YES -> IF	IGNORE	Train #4 is running.
5	TRUE	YES -> IF	IGNORE	Train #5 is running.
6	TRUE	YES -> IF	IGNORE	Train #6 is running.
7	TRUE	YES -> IF	IGNORE	Train #7 is running.
8	TRUE	YES -> IF	IGNORE	Train #8 is running.
9	TRUE	NO	NO -> ELSE	Train #9 is not operational.
10	TRUE	NO	YES -> ELSE-IF	Train #10 begins running at noon.
11	TRUE	NO	NO -> ELSE	Train #11 is not operational.
12	TRUE	NO	YES -> ELSE-IF	Train #12 begins running at noon.
13	FALSE	STOP THE LOOP!		

ADDING TO OUR LIST OF SPECIAL TRAINS

Using && for unique conditions

1

✓

2

✓

3

✓

4

✓

5

✓

6

✓

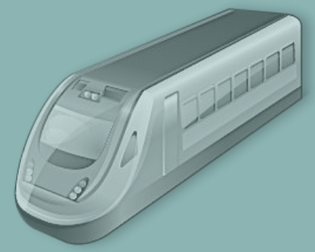
7

✓

8

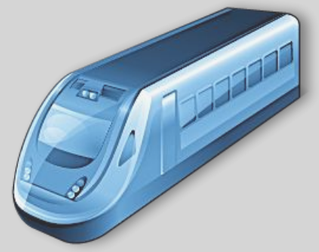
✓

9



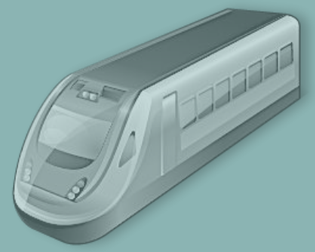
✗

10



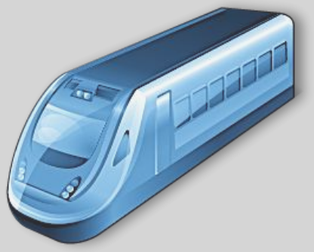
✓

11



✗

12



✓

ADDING TO OUR LIST OF SPECIAL TRAINS

Using && for unique conditions

1

✓

2

✓

4

✓

5

✓

6

✓

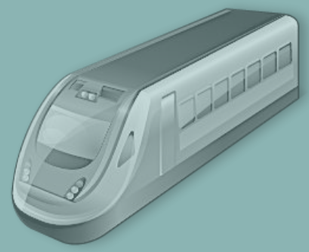
7

✓

8

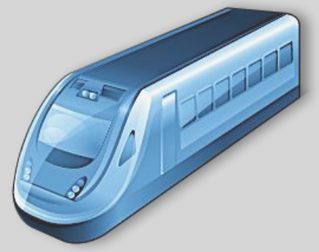
✓

9



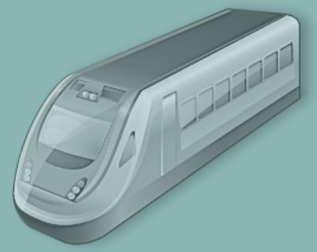
✗

10



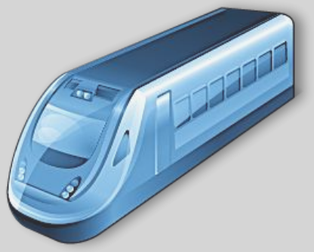
✓

11



✗

12



✓

ADDING TO OUR LIST OF SPECIAL TRAINS

Using && for unique conditions

1

✓

2

✓

3

✓

4

✓

5

✓

6

✓

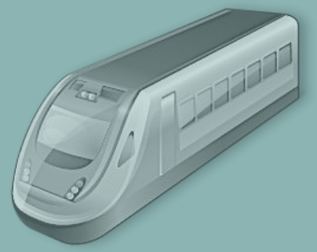
7

✓

8

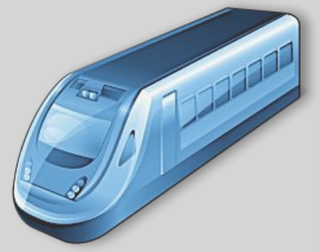
✓

9



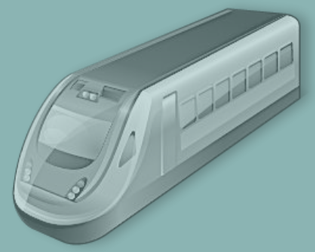
✗

10



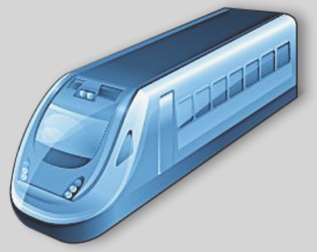
✓

11



✗

12



✓

INSERTING NEW CONDITIONS

We want to make sure Train 3 runs only on Sunday

```
var dayOfWeek = "Friday";
```

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains) {
            console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10 || trainNumber == 12 ) {
            console.log("Train # " + trainNumber + " will begin running at noon.");
     } else if (    *trainNumber is 3 AND its Sunday*    ) {
           *print that train 3 is running*
     } else {
            console.log("Train #" + trainNumber + " is not operational.");
     }
}
```

INSERTING NEW CONDITIONS

We want to make sure Train 3 runs only on Sunday

```
var dayOfWeek = "Friday";
```

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains) {
            console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10 || trainNumber == 12 ) {
            console.log("Train # " + trainNumber + " will begin running at noon.");
     } else if ( trainNumber == 3 && dayOfWeek == "Sunday"  ) {
            console.log("Train #3 is running.");
     } else {
            console.log("Train #" + trainNumber + " is not operational.");
     }
}
```

LETS RUN IT!

```
Train #1 is running.
Train #2 is running.
Train #3 is running.
Train #4 is running.
Train #5 is running.
Train #6 is running.
Train #7 is running.
Train #8 is running.
Train #9 is not operational.
Train #10 will begin running at noon.
Train #11 is not operational.
Train #12 will begin running at noon.
```

WHY DIDN'T WE GET THE RIGHT STATUS?

Tracing our loop logic

```
var dayOfWeek = "Friday";
var totalTrains = 12;
var trainsOperational = 8;
```

```
trainsOperational > 0
```

✓

```
trainsOperational == totalTrains
```

✗

```
trainNumber = 3
```

In our status loop, trainNumber eventually becomes 3

```
trainNumber <= operationalTrains
```

✓

→ Train #3 is running. ✗

But it shouldn't be running, because it's Friday!

HOUSTON, WE HAVE A PROBLEM...

Our logic doesn't work! What do we need...?

```
var dayOfWeek = "Friday";
```

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
    if (trainNumber <= operationalTrains) {
            console.log("Train #" + trainNumber + " is running.");
    } else if (trainNumber == 10 || trainNumber == 12 ) {
            console.log("Train # " + trainNumber + " will begin running at noon.");
    } else if ( trainNumber == 3 && dayOfWeek == "Sunday" ) {
            console.log("Train #3 is running.");
    } else {
            console.log("Train #" + trainNumber + " is not operational.");
    }
}
```

If there are 3 or more operational trains, then the later Else-If will never be checked when the trainNumber is 3. Thus, our system says that Train 3 is running when it isn't!

HOUSTON, WE HAVE A PROBLEM...

Our logic doesn't work! What do we need...?

```
var dayOfWeek = "Friday";
```

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains  *AND trainNumber is NOT 3* ) {
               console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10 || trainNumber == 12 ) {
               console.log("Train # " + trainNumber + " will begin running at noon.");
     } else if ( trainNumber == 3 && dayOfWeek == "Sunday" ) {
               console.log("Train #3 is running.");
     } else {
               console.log("Train #" + trainNumber + " is not operational.");
     }


}
```

HOUSTON, WE HAVE A PROBLEM...

Our logic doesn't work! What do we need...?

```
var dayOfWeek = "Friday";
```

trains.js

```
...
for (trainNumber = 1; trainNumber <= totalTrains; trainNumber++) {
     if (trainNumber <= operationalTrains && trainNumber != 3       ) {
               console.log("Train #" + trainNumber + " is running.");
     } else if (trainNumber == 10 || trainNumber == 12 ) {
               console.log("Train # " + trainNumber + " will begin running at noon.");
     } else if ( trainNumber == 3 && dayOfWeek == "Sunday" ) {
               console.log("Train #3 is running.");
     } else {
               console.log("Train #" + trainNumber + " is not operational.");
     }
}
```

If we have made sure that trainNumber is NOT 3 before printing out for a regular running train, this later Else-If will trigger correctly if both conditions are met!

NOW WE GET CORRECT PRINTOUTS!

```
var dayOfWeek = "Friday";
```

```
Train #1 is running.
Train #2 is running.
Train #3 is not operational.
Train #4 is running.
Train #5 is running.
Train #6 is running.
Train #7 is running.
Train #8 is running.
Train #9 is not operational.
Train #10 will begin running at noon.
Train #11 is not operational.
Train #12 will begin running at noon.
```

NOW WE GET CORRECT PRINTOUTS!

```
var dayOfWeek = "Sunday";
```

```
Train #1 is running.
Train #2 is running.
Train #3 is running.
Train #4 is running.
Train #5 is running.
Train #6 is running.
Train #7 is running.
Train #8 is running.
Train #9 is not operational.
Train #10 will begin running at noon.
Train #11 is not operational.
Train #12 will begin running at noon.
```

TRACING OUR COMPLEX CONDITIONAL

How do we arrive at the different printouts for Train 3?

```
var dayOfWeek = "Friday";
var totalTrains = 12;
var trainsOperational = 8;
```

```
trainsOperational > 0
```

```
trainsOperational == totalTrains
```

```
trainNumber = 3
```

In the loop, trainNumber eventually becomes 3

```
trainNumber <= operationalTrains && trainNumber != 3
```

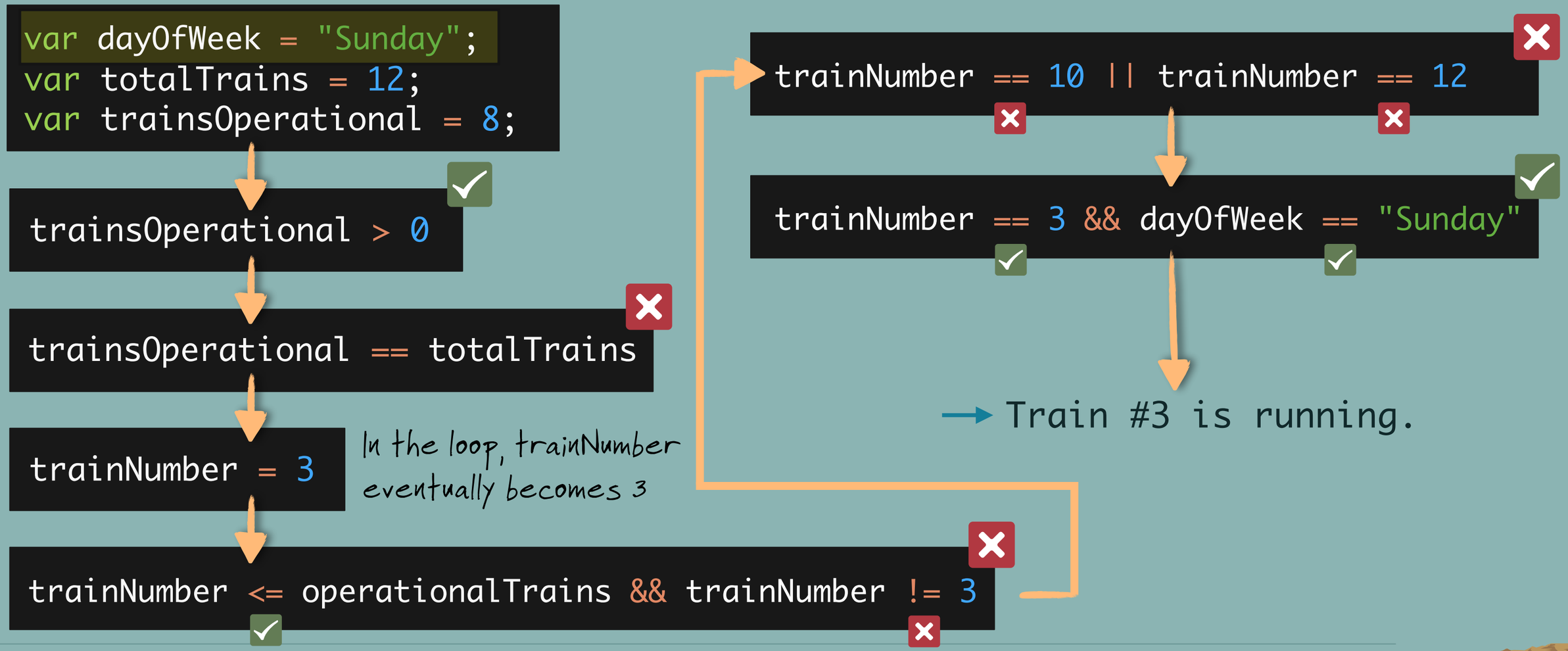
```
trainNumber == 10 || trainNumber == 12
```

```
trainNumber == 3 && dayOfWeek == "Sunday"
```

→ Train #3 is not operational.

TRACING OUR COMPLEX CONDITIONAL

How do we arrive at the different printouts for Train 3?




See the top of the world at
BUILT-IN'S BLUFF

LEVEL 3

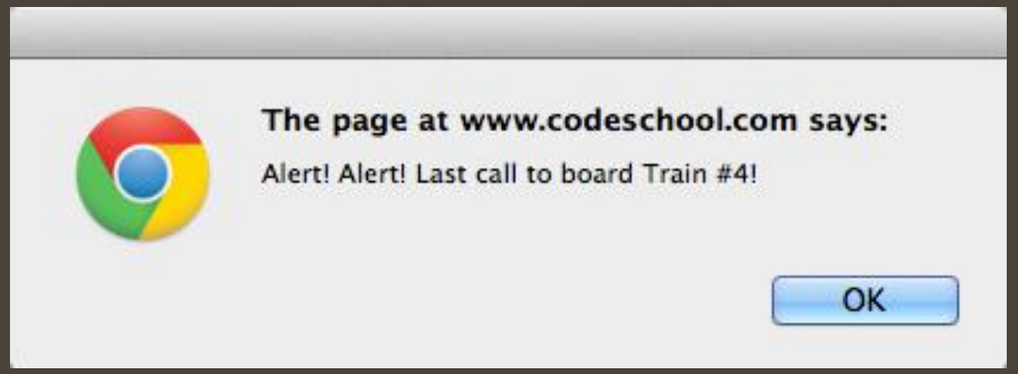
BUILT-INS BLUFF

EXISTING JAVASCRIPT FUNCTIONS

Built-in functions you can use at any time to get and send information

alert( ) Sends a message to the user in a small pop-up window

```
alert("Alert! Alert! Last call to board Train #4!");
```



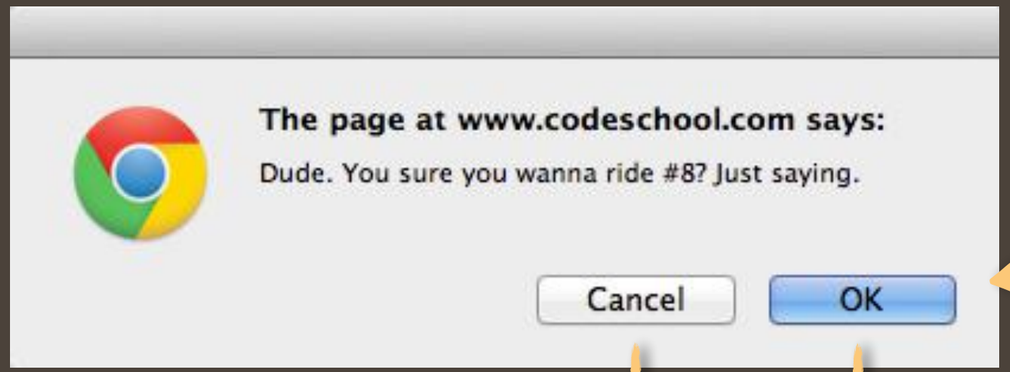
We send the message as a "parameter" to the function by enclosing it in parentheses. It can be any value or String.

EXISTING JAVASCRIPT FUNCTIONS

Built-in functions you can use at any time to get and send information

confirm( ) Asks user for consent to move forward with an action

```
confirm("Dude. Are you sure you wanna ride #8? Just saying.");
```



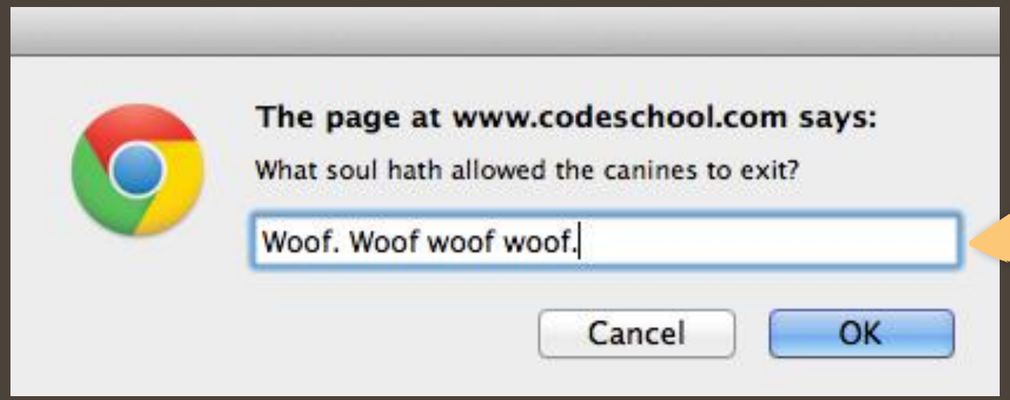
If the user hits OK, the confirm( ) function returns true. If the user hits cancel, the confirm( ) function returns false.

EXISTING JAVASCRIPT FUNCTIONS

Built-in functions you can use at any time to get and send information

prompt( ) Sends a message and retrieves an entry from the user

```
prompt("What soul hath allowed the canines to exit?");
```



"When the party was nice,
the party was jumping."
-William Shakespeare

USING DIALOGS WITH VARIABLES

Requesting information and storing with prompt( )

```
var userName = prompt("Yo passenger! What's your name?");
```

The prompt( ) function returns Gregg's entry, which is then stored in the variable userName.


```
> userName
```

→ "Gregg Pollack"

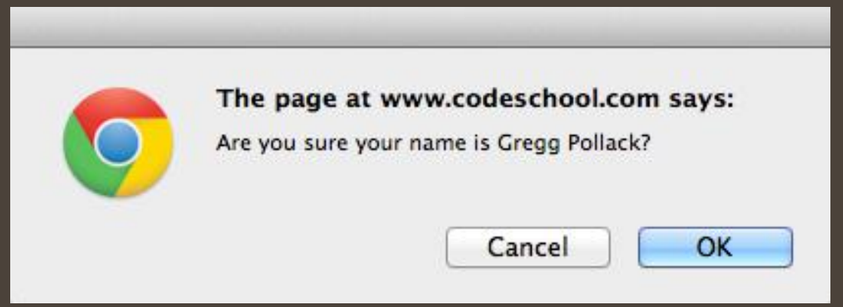
CONFIRMING OUR PASSENGER'S NAME

Using confirm( ) to ensure the user is satisfied with their entry

```
var userName = prompt("Yo passenger! What's your name?");
```


```
confirm("Are you sure your name is " + userName + "?");
```

But wait a minute. This doesn't really do anything. What if Gregg selects cancel, in either dialog box?



USING THE TYPEOF OPERATOR

Identifying the “type” of value inside a variable or expression

```
> typeof true
```

→ `"boolean"`

```
> typeof "That's not a valid entry!"
```

→ `"string"`

```
> typeof 42
```

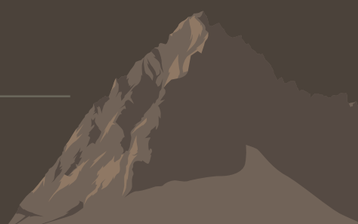
→ `"number"`

```
> typeof undefined
```

→ `"undefined"`

```
> typeof null
```

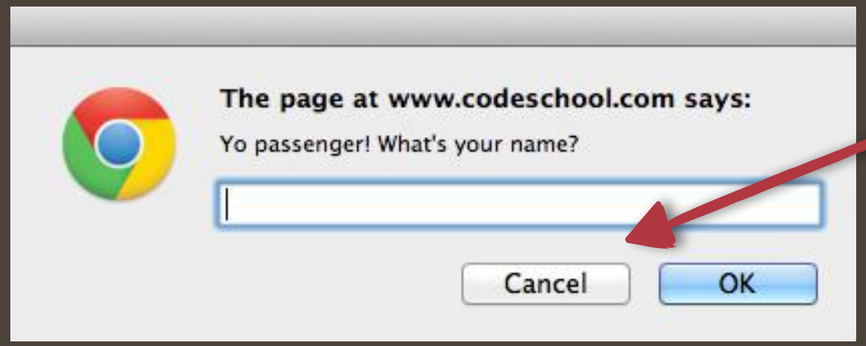
→ `"object"`



IDENTIFYING A USER'S RESPONSE

The typeof( ) method is useful in checking a variable's contents

```
var userName = prompt("Yo passenger! What's your name?");
```



If the user selects cancel without entering anything, prompt( ) will return a special value called "null," which is not a String.

```
> typeof userName
"object"
```

If userName is set to "null" by prompt( ), then it will not be a String, but instead, a generic JavaScript "object."

CANCELING A CONFIRM DIALOG BOX

Using a cancel to divert or restart a process

```
confirm("Are you sure your name is " + userName + "?");
```

Since confirm returns a true or false value, we can use it in conditionals!

```
if ( confirm(*user selects OK here*)  ) {

      *do some code, yo!*
}
```

A CONFIRMATION LOOP

Let's write some pseudo-code that plans our solution using functions

trains.js

```
...
*make a confirmation flag*
*until the user has confirmed a name, do this:*{
    *request the name using prompt()*

    *if the user says OK at confirm()*{
        *acknowledge the accepted entry*
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
*make a confirmation flag*
*until the user has confirmed a name, do this:*{
    *request the name using prompt()*

    *if the user says OK at confirm()*{
        *acknowledge the accepted entry*
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
*until the user has confirmed a name, do this:*{
    *request the name using prompt()*

    *if the user says OK at confirm()*{
        *acknowledge the accepted entry*
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

This 'flag' will control our loop, based on whether we've got the user's correct name yet.

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
while(gotName == false){
    *request the name using prompt()*

    *if the user says OK at confirm()*{
        *acknowledge the accepted entry*
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
while(gotName == false){
    var userName = prompt("Yo passenger! What's your name?");

    *if the user says OK at confirm()*{
        *acknowledge the accepted entry*
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
while(gotName == false){
    var userName = prompt("Yo passenger! What's your name?");

    if ( confirm("Are you sure your name is " + userName + "?") ){
        *acknowledge the accepted entry*
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
while(gotName == false){
    var userName = prompt("Yo passenger! What's your name?");

    if ( confirm("Are you sure your name is " + userName + "?") ){
        alert("'Sup " + userName + "!");
        *adjust flag to exit the loop*
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
while(gotName == false){
    var userName = prompt("Yo passenger! What's your name?");

    if ( confirm("Are you sure your name is " + userName + "?") ){
        alert("'Sup " + userName + "!");
        gotName = true;
    }

    *otherwise, cycle back to the top*


}
```

A CONFIRMATION LOOP

Now, we'll create code that matches our intent

trains.js

```
...
var gotName = false;
while(gotName == false){
    var userName = prompt("Yo passenger! What's your name?");

    if ( confirm("Are you sure your name is " + userName + "?") ){
        alert("'Sup " + userName + "!");
        gotName = true;
    }

}
```

If this statement never executes, the loop will restart!

A CONFIRMATION LOOP

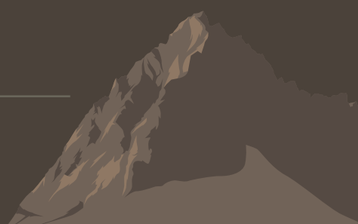
Now, we'll create code that matches our intent

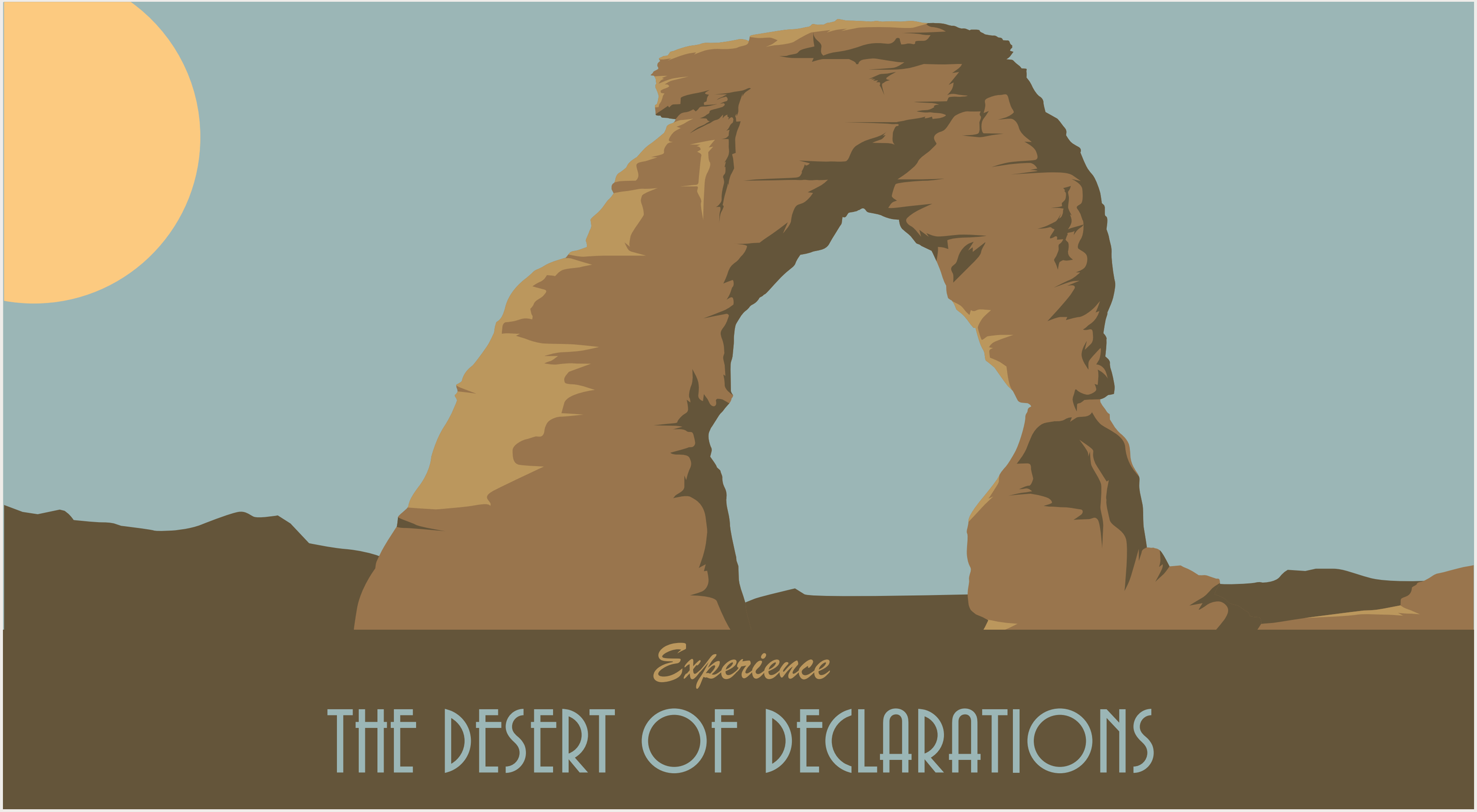
trains.js

```
...
var gotName = false;
while(gotName == false){
    var userName = prompt("Yo passenger! What's your name?");

    if ( confirm("Are you sure your name is " + userName + "?") ){
        alert("'Sup " + userName + "!");
        gotName = true;
    }


}
```

Now to see it in action!

CHECK OUT THE LAST VIDEO OF THIS LEVEL TO REVIEW THE SCREENCAST OF THIS EXECUTION!

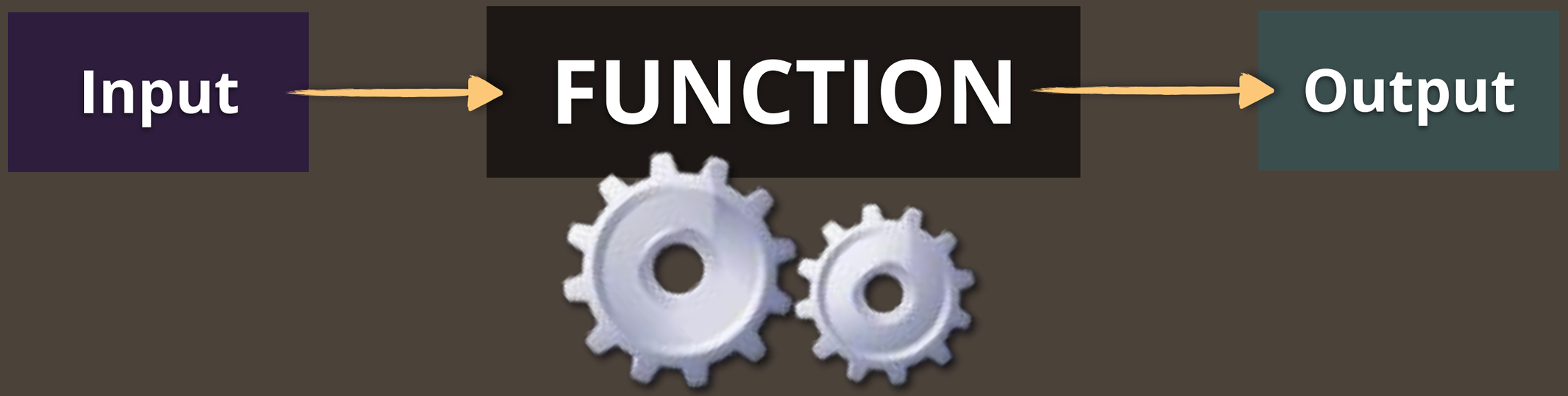



Experience
THE DESERT OF DECLARATIONS

LEVEL 4

THE DESERT OF DECLARATIONS

WHAT'S A FUNCTION FOR?



Give the function some input...

...it does some stuff to or with the input...

...and it outputs some result.

FUNCTIONS SOLVE PROBLEMS

A function "does something" step-by-step that we need to do repeatedly

FUNCTION: The Sum of Two Cubes

1. Get two numbers — 4 9
2. Cube each number — $4^3 = 64$ $9^3 = 729$
3. Sum the cubes — $64 + 729 = 793$
4. Return the answer — **793**

WHAT ARE THESE STEPS IN CODE?

Syntax for finding a sum of cubes

4 → `var a = 4;`

9 → `var b = 9;`

$4^3 = 64$ → `var aCubed = a*a*a;`

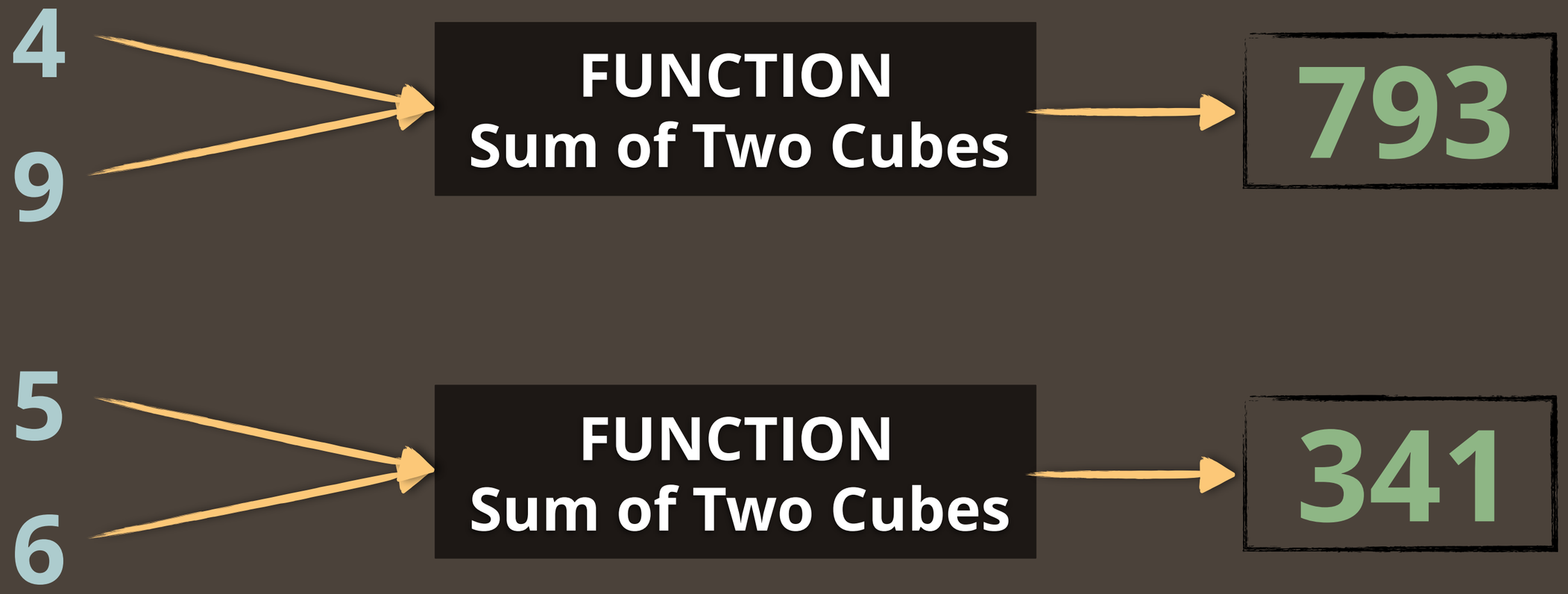
$9^3 = 729$ → `var bCubed = b*b*b;`

$64 + 729 = 793$ → `var sum = aCubed + bCubed;`

Without a function, we'd have to write this code a lot!

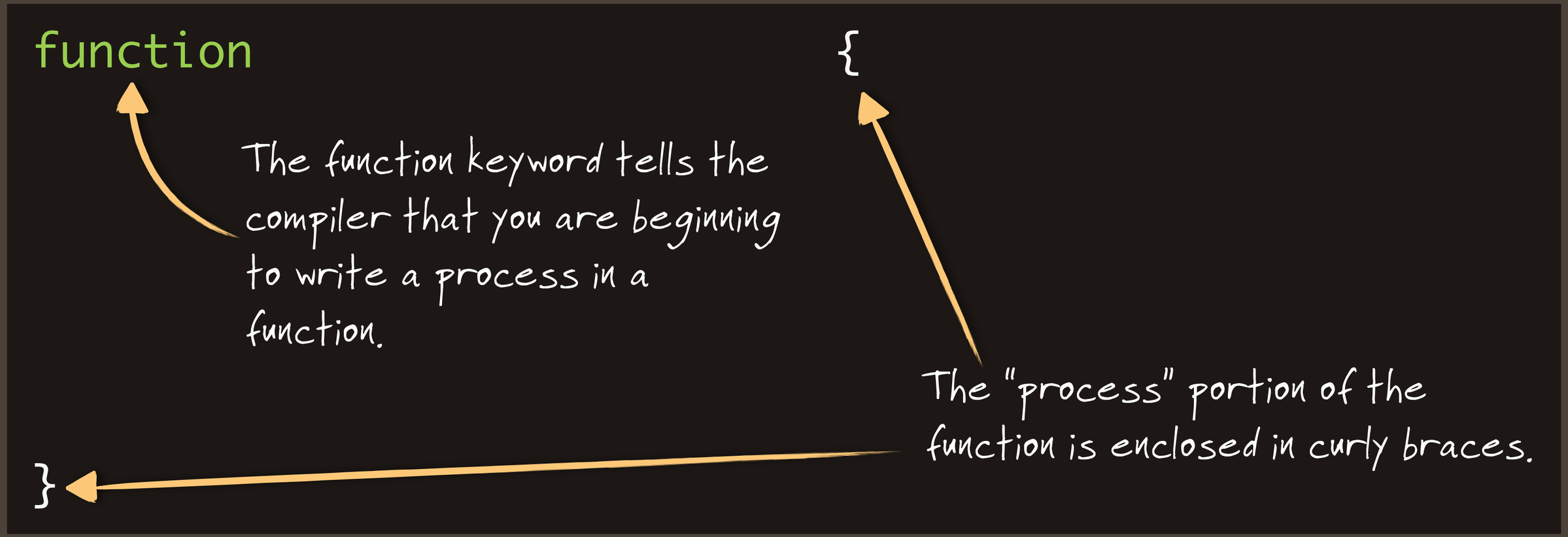
USEFULNESS THROUGH REUSABILITY

Wrapping our code in a function will allow us to reuse it



FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure



FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure

```
function  sumOfCubes  (a, b) {


}
```

The function's name follows the function keyword and should indicate briefly what's going on in the process.

Parameters are passed in a set of parentheses before the first curly brace. They are the "materials" the function will "work on".

FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure

```
function  sumOfCubes  (a, b) {

    *do some stuff*

    return *something (or nothing) from the process*
}
```

Inside the braces, the process occurs. In other words, the function does what it is intended to do.

This return keyword says to the function, "OK, we're done, now give us the result of what we did." It can be used anywhere in the function to stop the function's work. Here, that happens to be at the very end.

BUILDING OUR SUMOFCUBES FUNCTION

Assigning steps of the process to the function syntax

```
function  sumOfCubes  (a, b) {

      2. Cube each number

      3. Sum the cubes

    return  Sum

}
```

1. Get two numbers

4. Return the answer

BUILDING OUR SUMOFCUBES FUNCTION

Assigning steps of the process to the function syntax

```
function  sumOfCubes  (a, b)  {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;

    return sum;

}
```

Once the parameters are passed into the function, they are accessible at any point within the process.

CALLING OUR SUMOFCUBES FUNCTION

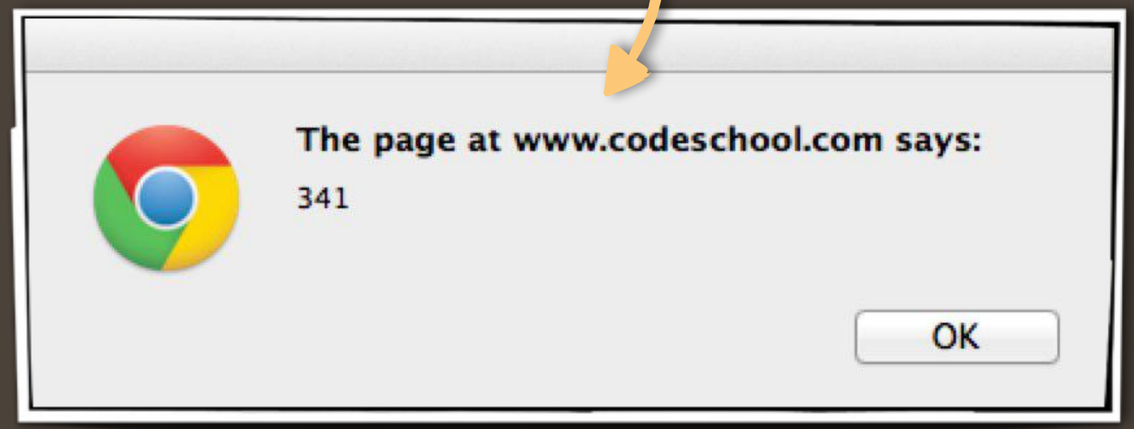
Now we can call the function using any parameter values we want!

```
function  sumOfCubes  (a, b)  {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;

    return sum;

}
```

```
sumOfCubes(4, 9);
```

→ 793

```
var mySum = sumOfCubes(5, 6);
alert(mySum);
```



WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

Our function does what it is supposed to, but it's not as efficient as it could be memory-wise. We've made three unnecessary variables that all have to be allocated in memory.

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    return aCubed + bCubed;
}
```

The return keyword can calculate the results of an expression before actually returning from the function. One variable down!

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    return aCubed + bCubed;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    return aCubed + b*b*b;
}
```

One more variable down! Why make a bCubed when we can just use the calculation as a substitute? You can guess, then, what's coming next.

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    return aCubed + bCubed;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    return aCubed + b*b*b;

}
```

```
function sumOfCubes(a, b) {

    return a*a*a + b*b*b;
}
```

Woohoo! One statement!

OUR FUNCTION IN ACTION

Calling a function involves the function name and some parameters

```
function sumOfCubes(a, b) {

    return a*a*a + b*b*b;
}
```

```
sumOfCubes(4, 9);
```

→ 793

Parameters can also be expressions, which the function will resolve before starting:

```
sumOfCubes(1+2, 3+5);
```

→ 539

Same as (3, 8)

```
var x = 3;
sumOfCubes(x*2, x*4);
```

→ 1494

Same as (6, 12)

NOW FOR A MORE COMPLEX FUNCTION!

Let's design a function that counts "E's" in a user-entered phrase

```
function countE ( ) {
      *ask user for a phrase to check*
      *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

      }




}
```

By the way, sometimes functions don't need any parameters!

Always check that a user input is valid before any operations

Using the return keyword here will allow us to exit and inform the program of an invalid entry.

NOW FOR A MORE COMPLEX FUNCTION!

Let's design a function that counts "E's" in a user-entered phrase

```
function countE ( ) {
      *ask user for a phrase to check*
      *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

      }*otherwise*{

      }
}
```

This block will be where the function begins to actually check the phrase out and count the E's.

NOW FOR A MORE COMPLEX FUNCTION!

Let's design a function that counts "E's" in a user-entered phrase

```
function countE ( ) {
      *ask user for a phrase to check*
      *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

      }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                  *if the character is an 'E' or an 'e'*{
                        *increment the E counter*
                  }
            }
            *alert the amount of E's in the phrase and return success*
      }
}
```

We have to count lowercase as well as uppercase!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
      *ask user for a phrase to check*
      *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

      }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                  *if the character is an 'E' or an 'e'*{
                        *increment the E counter*
                  }
            }
            *alert the amount of E's in the phrase and return success*
      }
}
```

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     *if the entry is invalid*{
          *alert the user*
          *exit function with a failure report*

     }*otherwise*{
          *make a counter for the E's*
          *for each character in the user's entry*{
               *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
               }
          }
          *alert the amount of E's in the phrase and return success*
     }
}
```

The prompt() method helps us get the user's entry.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {

     }
       *otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*
     }
}
```

The typeof keyword allows us to determine whether the user has entered a valid string. This != expression returns true or false.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
             alert("That's not a valid entry!");
             return false;
     }
       *otherwise*{
             *make a counter for the E's*
             *for each character in the user's entry*{
                  *if the character is an 'E' or an 'e'*{
                       *increment the E counter*
                  }
             }
             *alert the amount of E's in the phrase and return success*
     }
}
```

If the entry is not a string, we alert the user and exit the function, returning false.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
             alert("That's not a valid entry!");
             return false;
     } else {

             *make a counter for the E's*
             *for each character in the user's entry*{
                 *if the character is an 'E' or an 'e'*{
                     *increment the E counter*
                 }
             }
             *alert the amount of E's in the phrase and return success*
     }
}
```

Else-blocks help us do the "otherwise" case!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
              alert("That's not a valid entry!");
              return false;
     } else {

              var eCount = 0;
              for (var index = 0; index < phrase.length; index++) {

                   *if the character is an 'E' or an 'e'*{
                        *increment the E counter*
                   }
              }
              *alert the amount of E's in the phrase and return success*

     }
}
```

We want to start at index 0, and go until one less than the length of the user's string. Remember that strings have zero-based indices!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
             alert("That's not a valid entry!");
             return false;
     } else {

             var eCount = 0;
             for (var index = 0; index < phrase.length; index++) {
                 if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E')
                     eCount++;

                 }
             }
             *alert the amount of E's in the phrase and return success*
     }
}
```

This complex conditional checks whether the spot we're currently at along the string is either an `E` or an `e`.

If we found one, we'll increase our counter.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
              alert("That's not a valid entry!");
              return false;
     } else {

              var eCount = 0;
              for (var index = 0; index < phrase.length; index++) {
                   if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E')
                        eCount++;
                   }
              }

              *alert the amount of E's in the phrase and return success*

     }
}
```

This complex conditional checks whether the spot we're currently at along the string is either an E or an e.

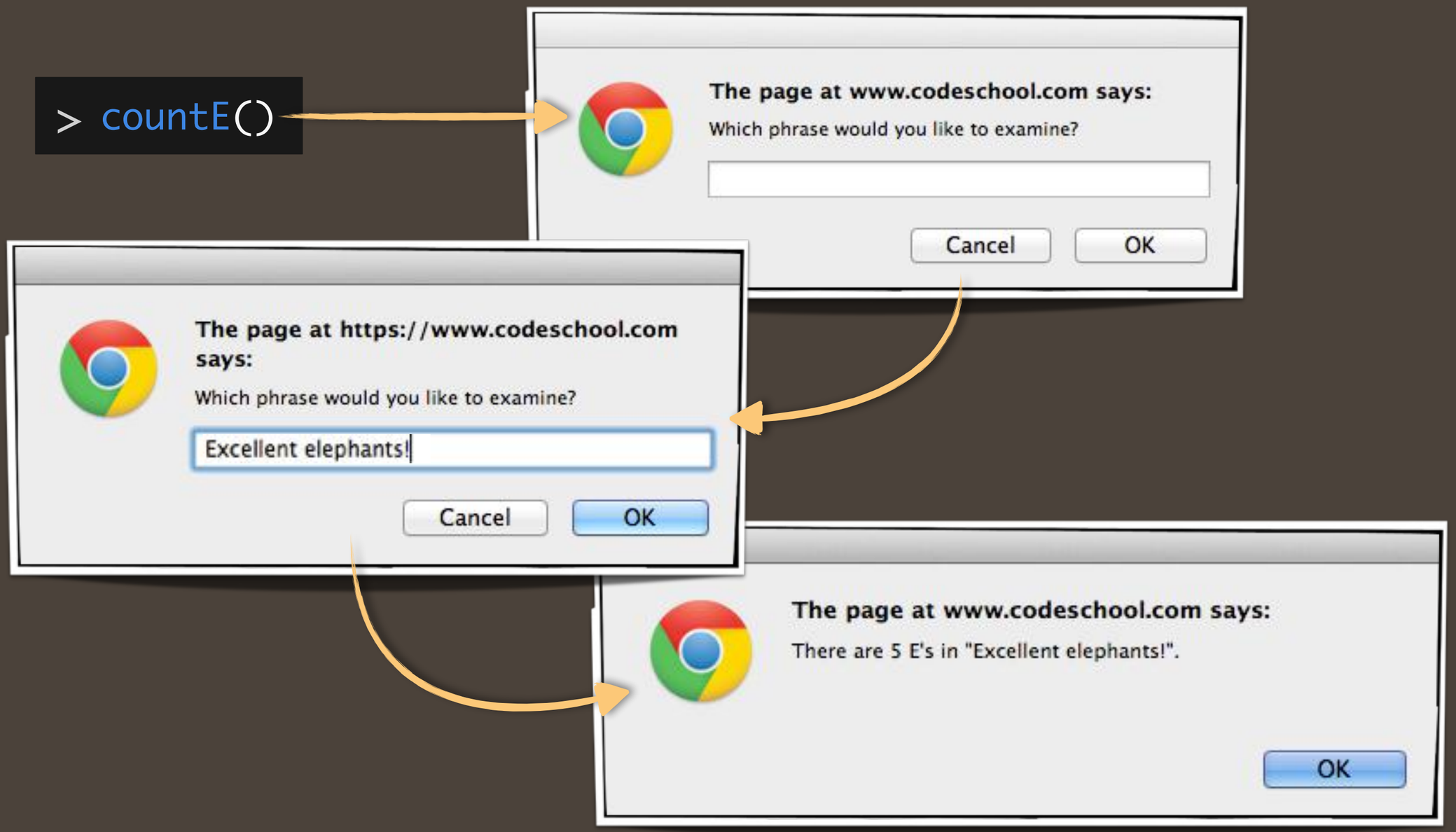
FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
               alert("That's not a valid entry!");
               return false;
     } else {

               var eCount = 0;
               for (var index = 0; index < phrase.length; index++) {
                    if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E')
                         eCount++;
                    }
               }
               alert("There are " + eCount + " E's in \"" + phrase + "\".");
               return true;
     }
}
```

After our for loop, eCount will contain the total number of E's and e's in our loop.

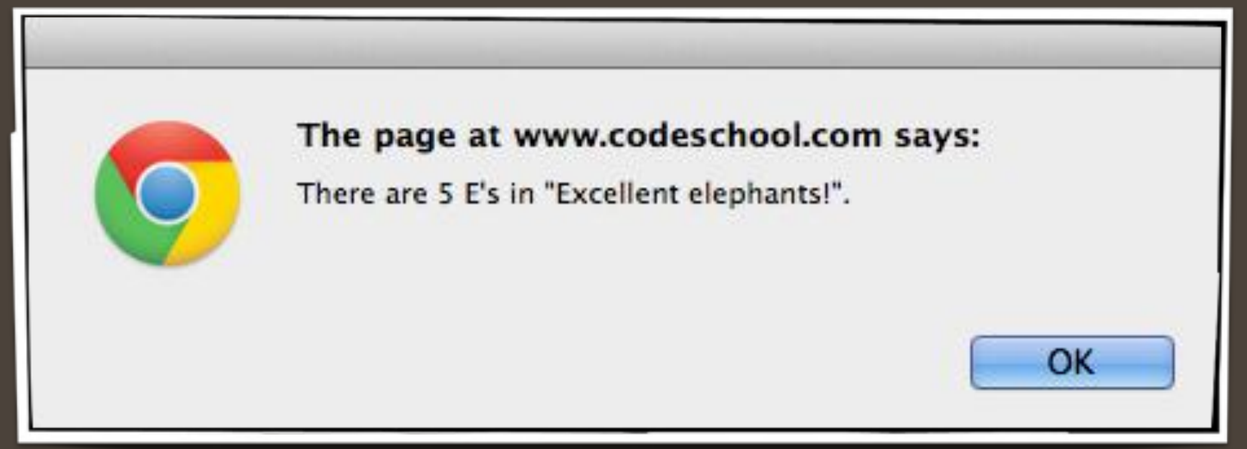
THE SEQUENCE OF ENTRY



TRACING OUR E-COUNTER

Following our function's code as it counts E's in "Excellent elephants!"

index	LOOP: index < length?	charAt (index)	is charAt(index) an E or e?	eCount
0	TRUE	E	TRUE	1
1	TRUE	x	FALSE	1
2	TRUE	c	FALSE	1
3	TRUE	e	TRUE	2
4	TRUE	l	FALSE	2
5	TRUE	l	FALSE	2
6	TRUE	e	TRUE	3
7	TRUE	n	FALSE	3
8	TRUE	t	FALSE	3
9	TRUE	(space)	FALSE	3
10	TRUE	E	TRUE	4
11	TRUE	l	FALSE	4
12	TRUE	e	TRUE	5
13	TRUE	p	FALSE	5
14	TRUE	h	FALSE	5
15	TRUE	a	FALSE	5
16	TRUE	n	FALSE	5
17	TRUE	t	FALSE	5
18	TRUE	s	FALSE	5
19	TRUE	!	FALSE	5
20	FALSE	STOP!		



UNDERSTANDING LOCAL AND GLOBAL SCOPE

Visualizing worlds within worlds...

```
var x = 6;
var y = 4;

function add (a, b){

    var x = a + b;
    return x;
}

function subtract (a, b){

    y = a - b;
    return y;
}
```

Out here, in the main program, the scope is "global", which means that variables declared are potentially accessible from everywhere.

Inside functions, the scope is "local", like cities within a state. Each has their own "government" and stuff that happens in here stays in here.

FUNCTIONS CREATE A NEW SCOPE

Variables declared in a function STAY in the function

```
var x = 6
function add (a, b){

    var x = a + b;
    return x;
}
```

```
add(9, 2);
```

→ 11

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

→ 6

The circled variable only exists in the function's local scope. Because it has been declared with var, it doesn't modify the same-named variable "outside" the function.

```
var x = 6
function add (a, b){

    x = a + b;
    return x;
}
```

```
add(9, 2);
```

→ 11

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

→ 11

If the x were not declared with var, it "shadows" the same-named variable from the nearest external scope!

VISUALIZING LOCAL AND GLOBAL SCOPE

Worlds within worlds...

PROGRAM

variables: x, y
functions: add, subtract

add

parameters:
a (local), b (local)

variables:
x (local)

subtract

parameters:
a (local), b (local)

variables:
y (GLOBAL)

```
var x = 6;
var y = 4;
function add (a, b){

    var x = a + b;
    return x;
}

function subtract (a, b){

    y = a - b;
    return y;
}
```



Getaway to
THE ARRAY ARCHIPELAGO

LEVEL 5

THE ARRAY ARCHIPELAGO

WHAT IF WE WANTED A PASSENGER LIST?

How would we structure a list of passengers inside our train.js system?

trains.js

```
...
function makeList ( ) {

    var passengerOne = "Gregg Pollack";
    var passengerTwo = "Aimee Simone";
    var passengerThree = "Thomas Meeks";
    var passengerFour = "Olivier Lacan";

    ...and on and on, typing through a list
    of sixty passengers, that might
    even change later?? No way.
}
...
```

THE ARRAY

An array is a data structure with automatically indexed positions

A 6-cell Array of Passengers

0 1 2 3 4 5

Just like Strings, Arrays have indices that are zero-based.

Despite his excellent disguise, it looks like Jon is in index 4. We mustache him a question.

ARRAY CELLS CAN HOLD ANY VALUE

Our picture array could also be an array of strings.

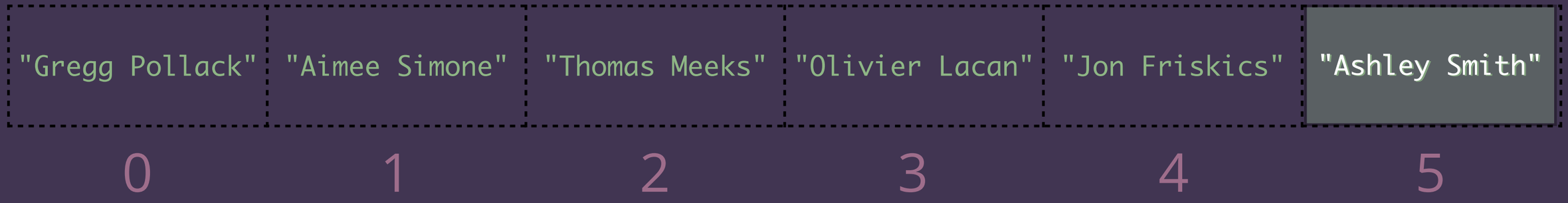
0 1 2 3 4 5

"Gregg Pollack"	"Aimee Simone"	"Thomas Meeks"	"Olivier Lacan"	"Jon Friskics"	"Ashley Smith"

0 1 2 3 4 5

BUILDING AND ACCESSING ARRAYS

Easy to build, easy to access with indices



To build this array in code, we write:

```
var passengers = [ "Gregg Pollack", "Aimee Simone", "Thomas Meeks",
                   "Olivier Lacan", "Jon Friskics", "Ashley Smith"];
```

The brackets indicate to the compiler to make an array and fill it with the comma-separated values between the brackets.

If we wanted to access any particular index's value, we use:

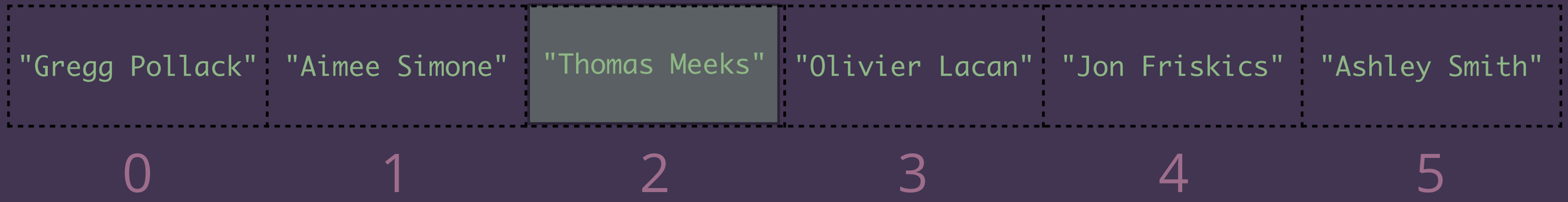
```
passengers[5];
```

→ `"Ashley Smith"`

Returns the value at index 5.

CHANGING ARRAY CONTENTS

We can also reference and change specific cells with indices



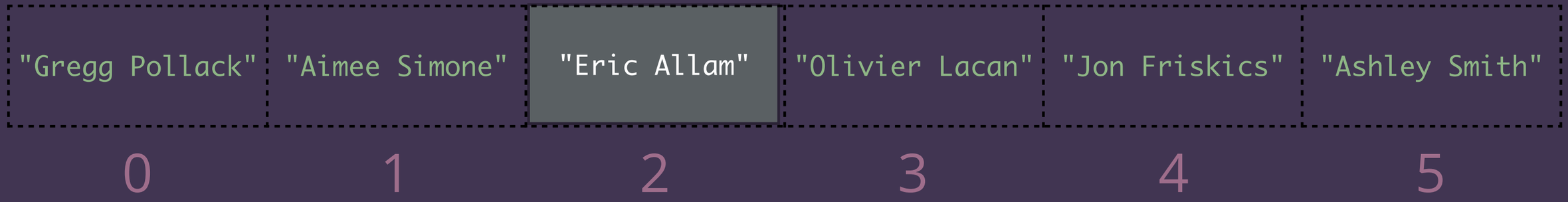
If we wanted to change the value contained at any index, we use:

```
passengers[2] = "Eric Allam";
```

This syntax says "Go over to index 2, and change its value to whatever comes after the = sign.

CHANGING ARRAY CONTENTS

We can also reference and change specific cells with indices



If we wanted to change the value contained at any index, we use:

```
passengers[2] = "Eric Allam";
```

This syntax says "Go over to index 2, and change its value to whatever comes after the = sign.

Like Strings, we can access the length of Arrays:

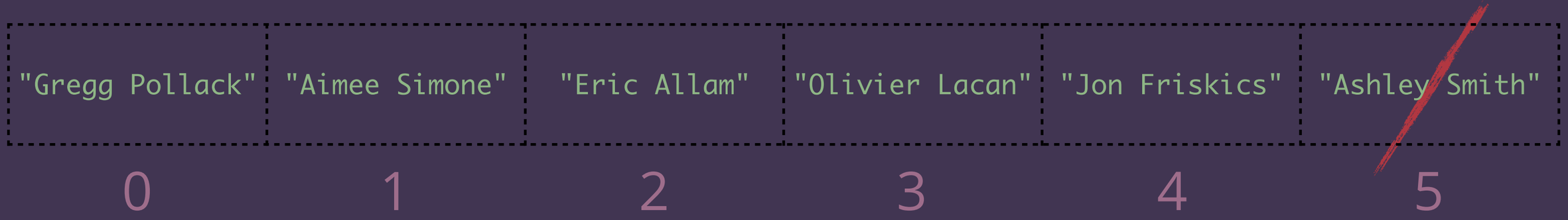
```
passengers.length;
```

→ 6

The length of an array is the actual number of cells, including any empty cells.

THE POP() FUNCTION

Removing a cell from the back of the array



The pop( ) function deletes the last position and retrieves its value:

```
passengers.pop();
```

→ `"Ashley Smith"`

pop() will automatically "pop" the last existing cell off the array while returning that cell's contents.

The array length automatically adjusts!

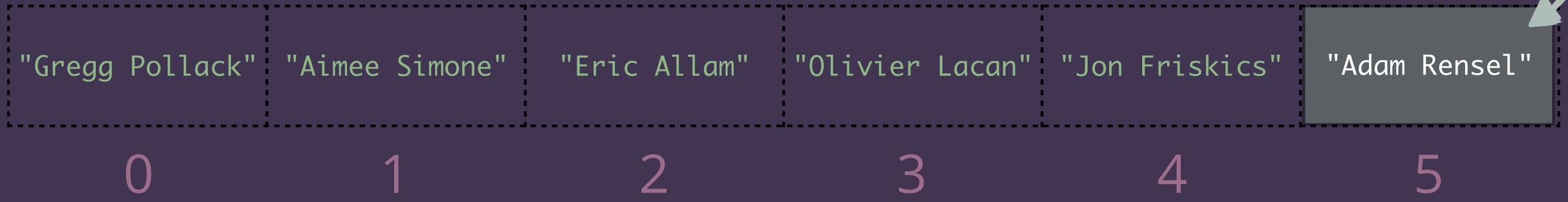
THE PUSH() FUNCTION

Adding a cell and its contents to the back of the array

The push( ) function adds a cell in the last position and enters a value:

```
passengers.push("Adam Rensel");
```

push() will "push" a cell onto the back of the array and automatically increase the array length.



ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var comboArray1 = ["One", "fish", 2, "fish"];
```

"One"	"fish"	2	"fish"
0	1	2	3

```
var poisson = "fish";
```

The variable name disappears in the array and just the contents remain.

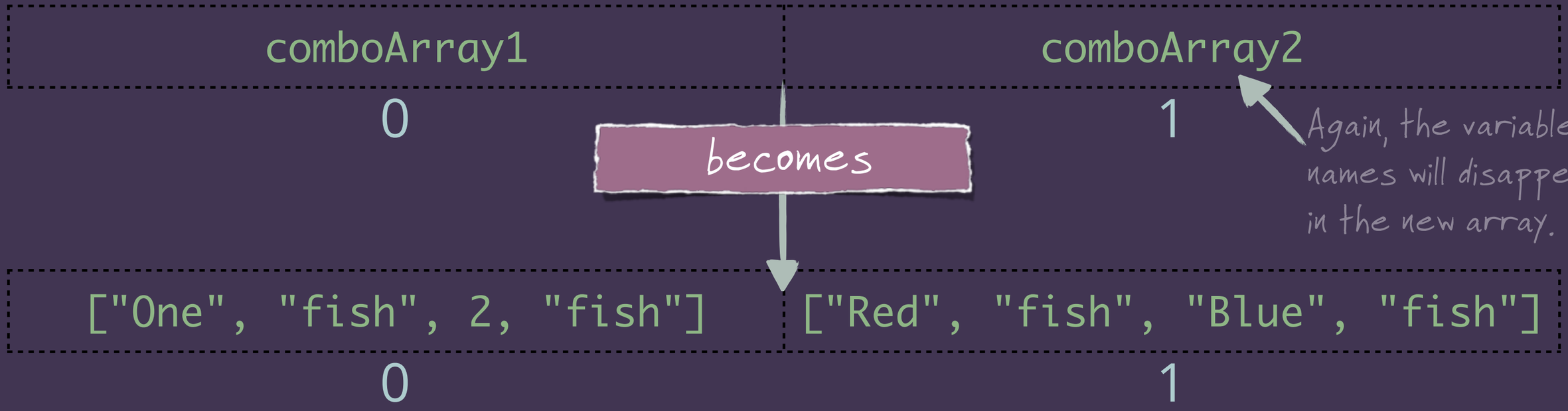
```
var comboArray2 = ["Red", poisson, "Blue", poisson];
```

"Red"	"fish"	"Blue"	"fish"
0	1	2	3

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var arrayOfArrays = [comboArray1, comboArray2];
```




```
console.log( arrayOfArrays );
```

→ [Array[4], Array[4]]

Here, the [4] and [4] are providing the lengths of each of the arrays, which here happen to be the same.

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var arrayOfArrays = [comboArray1, comboArray2];
```

```
["One", "fish", 2, "fish"]          ["Red", "fish", "Blue", "fish"]
```

0 1

```
console.log( arrayOfArrays[1] );
```

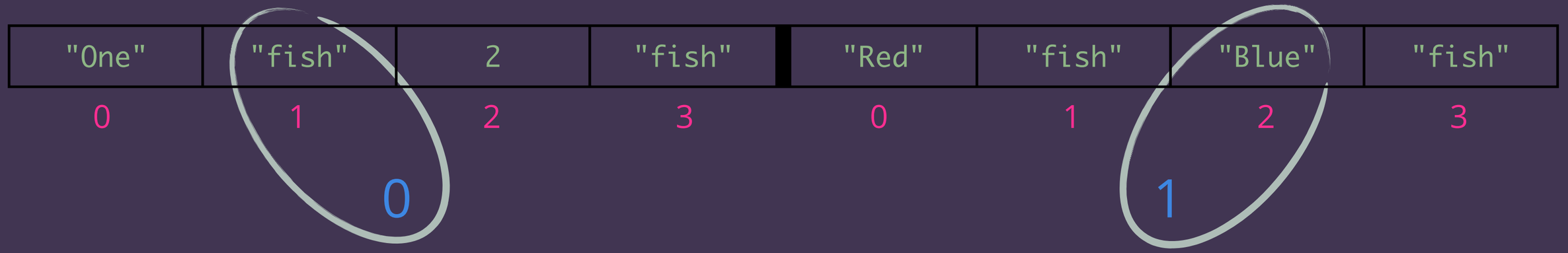
→ `["Red", "fish", "Blue", "fish"]`

When we reference the `[1]` index of `arrayOfArrays`, we get another entire array because that's what the cell contains. Specifically, our earlier `comboArray2`.

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var arrayOfArrays = [comboArray1, comboArray2];
```



The first bracket selects a cell in the master array.

The second bracket then selects a cell in the lower level array

```
console.log( arrayOfArrays[1][2] );
```

→ Blue

```
console.log( arrayOfArrays[0][1] );
```

→ fish

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
for (var i = 0; i < numberList.length; i++){
}
```

You'll often see the variable `i` used as a loop counter by convention and for simplicity.

To look through our entire array, we continue only until we have reached the last index of the zero-based array. Since our array has a `length` of `10`, we want to stop checking at index `9`.

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
for (var i = 0; i < numberList.length; i++){

  console.log("The value in cell " + i + " is " + numberList[i]);

}
```

Our loop counter can also serve as a current index position, helping us "iterate" over the entire contents of the array in order.

Don't confuse the index number (the *position*) with the contents of the cell (the *value*)!

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

i	i < numberList.length ?	numberList[i]	printout
0	TRUE	2	The value in cell 0 is 2
1	TRUE	5	The value in cell 1 is 5
2	TRUE	8	The value in cell 2 is 8
3	TRUE	4	The value in cell 3 is 4
4	TRUE	7	The value in cell 4 is 7
5	TRUE	12	The value in cell 5 is 12
6	TRUE	6	The value in cell 6 is 6
7	TRUE	9	The value in cell 7 is 9
8	TRUE	3	The value in cell 8 is 3
9	TRUE	11	The value in cell 9 is 11
10	FALSE	NA	STOP!

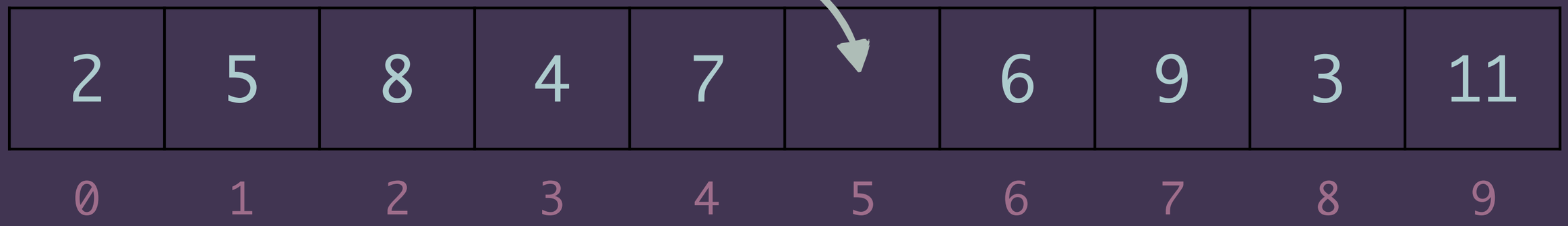
EMPTY CELLS IN ARRAYS?

Using the undefined value to create “empty” cells.

2	5	8	4	7	12	6	9	3	11
0	1	2	3	4	5	6	7	8	9

To make a cell empty, we’ll use the special `undefined` value, which means “no contents.”

```
passengers[5] = undefined;
```



A NEW FUNCTION WITH ARRAYS

Let's count even numbers AND erase odds.

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
var evenCount = 0;
for (var i = 0; i < numberList.length; i++) {
      if (numberList[i] % 2 == 0) {
            evenCount++;
      }


}
```

We'll set up a counter before the loop.

Even numbers will have a zero remainder when divided by 2!

A NEW FUNCTION WITH ARRAYS

Let's count even numbers AND erase odds.

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
var evenCount = 0;
for (var i = 0; i < numberList.length; i++) {
     if (numberList[i] % 2 == 0) {
          evenCount++;
     } else {
          numberList[i] = undefined;
     }
}
```

Otherwise, if not's even, we know it's odd! Here's where we will use undefined.

```
console.log(evenCount);
```

→ 5

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

i	i < numberList.length ?	numberList[i]	numberList[i] % 2 == 0 ?	evenCount
0	TRUE	2	**TRUE**	**1**
1	TRUE	5	FALSE	1
2	TRUE	8	**TRUE**	**2**
3	TRUE	4	**TRUE**	**3**
4	TRUE	7	FALSE	3
5	TRUE	12	**TRUE**	**4**
6	TRUE	6	**TRUE**	**5**
7	TRUE	9	FALSE	5
8	TRUE	3	FALSE	5
9	TRUE	11	FALSE	5
10	FALSE	NA	STOP!	

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

```
console.log(numberList);
```

→ `[2, undefined, 8, 4, undefined, 12, 6, undefined, undefined, undefined]`


```
console.log(numberList.length);
```

→ `10`

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( *passenger's name*, *array of passengers*) {
        *if list is empty* {
            *add passenger to list*
        } *else* {
            *for all spots in the list*{
                *if the current spot is empty* {
                    *add passenger to that spot*
                    *return the list and exit the function*
                } *else, if the end of the list is reached* {
                    *add passenger to end of list*
                    *return the list and exit the function*
                }
            }
        }
}
```

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {
        if (list.length == 0) {
            *add passenger to list*
        } *else* {
            *for all spots in the list*{
                    *if the current spot is empty* {
                            *add passenger to that spot*
                            *return the list and exit the function*
                    } *else, if the end of the list is reached* {
                            *add passenger to end of list*
                            *return the list and exit the function*
                    }
            }
        }
}
```

A length of 0 means the array is empty.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
          *for all spots in the list*{
              *if the current spot is empty* {
                  *add passenger to that spot*
                  *return the list and exit the function*
              } *else, if the end of the list is reached* {
                  *add passenger to end of list*
                  *return the list and exit the function*
              }
          }
      }
}
```

We start the list by pushing a passenger into the empty array.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
          for (var i = 0; i < list.length; i++) {
              *if the current spot is empty* {
                  *add passenger to that spot*
                  *return the list and exit the function*
              } *else, if the end of the list is reached* {
                  *add passenger to end of list*
                  *return the list and exit the function*
              }
          }
      }
}
```

We want to check all spots in the list, which will include all indices through `list.length - 1`

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
           for (var i = 0; i < list.length; i++) {
                if(list[i] == undefined){
                     list[i] = name;
                      *return the list and exit the function*
                } *else, if the end of the list is reached* {
                      *add passenger to end of list*
                      *return the list and exit the function*
                }
           }
      }
}
```

If a passenger spot has been emptied, it will be `undefined`. We want to fill that empty spot before adding more spots to the list.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
           for (var i = 0; i < list.length; i++) {
                if(list[i] == undefined){
                     list[i] = name;
                     return list;
                } *else, if the end of the list is reached* {
                      *add passenger to end of list*
                      *return the list and exit the function*
                }
           }
      }
}
```

If we've placed the passenger name, then we're done! No need to keep looping. We can now `return` the updated list and exit the function.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
          for (var i = 0; i < list.length; i++) {
              if(list[i] == undefined){
                  list[i] = name;
                  return list;
              } else if (i == list.length - 1) {
                  list.push(name);
                   *return the list and exit the function*
              }
          }
      }
}
```

If we have reached the final index of `list` without finding an empty spot, then push the name onto the end of `list`.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
          for (var i = 0; i < list.length; i++) {
              if(list[i] == undefined){
                  list[i] = name;
                  return list;
              } else if (i == list.length - 1) {
                  list.push(name);
                  return list;
              }
          }
      }
}
```

If the list was initially empty, we can return the updated `list` and exit.

CREATING A NEW PASSENGER LIST

Let's make a new list and add a few passengers to it.

```
var passengerList = [ ];
```

An empty set of brackets will create an array with no cells.

```
passengerList = addPassenger("Gregg Pollack", passengerList );
```

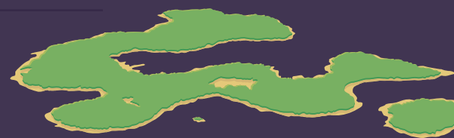
```
["Gregg Pollack"]
```

```
passengerList = addPassenger("Ashley Smith", passengerList );
```

```
["Gregg Pollack", "Ashley Smith"]
```

```
passengerList = addPassenger("Jon Friskics", passengerList );
```

```
["Gregg Pollack", "Ashley Smith", "Jon Friskics"]
```

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        }









}
```

If the list is empty, log it to the user.

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {









            }
        }
}
```

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {
                if(list[i] == name){
                    list[i] = undefined;
                }
        }
}
```

If the contents of the index match the name exactly, delete it by setting the index to undefined.

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {
                  if(list[i] == name){
                        list[i] = undefined;
                        return list;
            }
        }
}
```

Once we've deleted the passenger, we don't need any more loop cycles, so `return` will exit the entire function with the updated `list`.

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {
                if(list[i] == name){
                    list[i] = undefined;
                    return list;
                } else if (i == list.length - 1) {
                    console.log("Passenger not found!");
                }
            }
        }
}
```

If we get to the end, and we haven't deleted a name, then we know the passenger wasn't present!

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {
                if(list[i] == name){
                    list[i] = undefined;
                    return list;
                } else if (i == list.length - 1) {
                    console.log("Passenger not found!");
                }
            }
        }
        return list;
}
```

If the list was empty, or if we never found the passenger, we just return the same list.

MODIFYING OUR PASSENGER LIST

Let's take some passengers out, and put some back in.

```
passengerList = ["Gregg Pollack", "Ashley Smith", "Jon Friskics"];
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
["Gregg Pollack", undefined, "Jon Friskics" ]
```

```
passengerList = addPassenger( "Adam Rensel", passengerList );
```

```
["Gregg Pollack", "Adam Rensel", "Jon Friskics" ]
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
→ Passenger not found!
```

MODIFYING OUR PASSENGER LIST

Let's take some passengers out, and put some back in.

```
passengerList = ["Gregg Pollack", "Adam Rensel", "Jon Friskics"];
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
→ Passenger not found!
```

MODIFYING OUR PASSENGER LIST

Let's take some passengers out, and put some back in.

```
passengerList = ["Gregg Pollack", "Adam Rensel", "Jon Friskics"];
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

→ Passenger not found!

```
passengerList = deletePassenger("Gregg Pollack", passengerList );
```

[undefined, "Adam Rensel", "Jon Friskics"]

```
passengerList = addPassenger("Jennifer Borders", passengerList );
```

["Jennifer Borders", "Adam Rensel", "Jon Friskics"]

FIN?