

# Advanced Topics in vanilla JS



# Agenda

- ❖ setInterval
- ❖ setTimeout
- ❖ "use strict"
- ❖ Namespace
- ❖ preventDefault
- ❖ event propagation/bubbling
- ❖ pointer events
- ❖ scope
- ❖ closure
- ❖ Nested functions

# Learn how to learn - setInterval()

Take 2 minutes and find out what is setInterval.

## How do we do it?

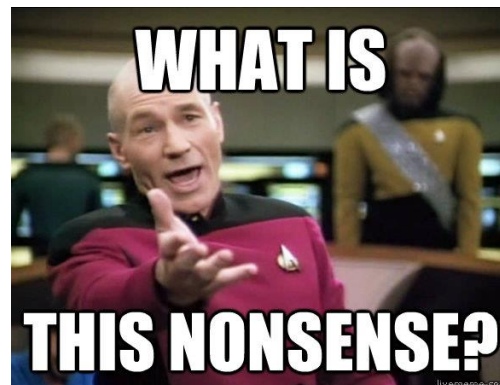
1. Read the description of the functionality of the function from a documentation site
2. Syntax: Function signature
  - Parameters + their type
  - Return type + description
3. Test the function: run it and see how does it behave
4. Look for a usage example online

# It's a matter of time

## Asynchronous code

JS has a magical ability of timing events:

- 10 seconds after loading the page, show a popup for email subscription
- Show the time (in digital form)
- Refresh the news feed every 30 seconds
- ...
- ▶ This is called asynchronous code: the action doesn't happen exactly after an event, but when the programmer chooses



# setInterval()

Do something every X milliseconds

```
setInterval(someCode, 1000);
```

Executed  
code



Time to  
repeat



```
var alertInterval = setInterval(function() {  
    alert("This is annoying");  
}, 5000)
```

Anonymous function



- ▶ This function throws an alert every 5 seconds

# setInterval()

Or use clean code:

```
var my_alert = function(){  
    alert("This is annoying");  
}  
  
var alertInterval = setInterval(my_alert, 5000);
```

- ▶ **No parameters:** Note that this can only be done if the function does not receive any parameters.
- ▶ **Async:** Note we are passing the function as an argument.  
The function will be executed only after 5 sec.

# Controlling setInterval()

## Clear Interval

When we want to stop a repeated event from happening again, we just need to use clearInterval function.

Don't forget to assign the setInterval return value into a variable  
So you can stop it when you want to

```
var alertIntervalId = setInterval(my_alert, 5000);  
clearInterval(alertIntervalId);
```

# setTimeout()

Do something (once) after waiting X milliseconds

```
setTimeout(someCode, 1000);
```

Executed  
code



Delay



```
console.log("This will print first");  
setTimeout(function () {  
    console.log("This will print third");  
}, 1000 );  
console.log("This will print second");
```

This will print first
This will print second
This will print third



# Understanding setTimeout()

When our code encounters a setTimeout function, it **Moves on** and executes what comes after



In parallel to that, JS counts down to when the timeout should end



When the timeout ends, the code inside the setTimeout function is being executed

# Controlling setTimeout()

When we want to stop a delayed event from happening, we just need to use `clearTimeout`

```
var alertTimeout = setTimeout(my_alert, 5000);  
clearTimeout(alertTimeout);
```

You can assign the `setTimeout` into a variable, so you can stop it from executing if you want to

# Questions

```
console.log("Questions?");
```

# Scope

In JavaScript there are two types of scope:

- Local scope
- Global scope

```
1  
2  
3 var global = 10;  
4  
5 function fun() {  
6  
7     var local = 5;  
8  
9 }  
10  
11  
12  
13
```

global variable

local variable

# Scope

JavaScript has function scope:  
Each function creates a new scope.

Scope determines the accessibility of these variables.

Variables defined inside a function are not accessible from outside the function.

# Scope

## Local scope vs Global scope example

```
//js file
```

```
// global scope
```

```
var global_var = 6;
```

```
function local_scope(){
```

```
  // local scope
```

```
  var local_var = "local"; // accessible only in the function
```

```
  console.log(global_var); // accessible everywhere
```

```
}
```

```
console.log(local_var);
```

```
//undefined. accessible only in the function
```

# Scope Example

Local scope vs Global scope example

```
var message = "hello";

function sayHello(name){
  message = "Hey";
  console.log(message + ' ' + name);
}

sayHello("Nana");
console.log(message);
```

What will be printed?

# Scope Example

Local scope vs Global scope example

```
var message = "hello";
```

```
function sayHello(name){  
  message = "Hey";  
  console.log(message + ' ' + name);  
}
```



Overriding the  
global variable

```
sayHello("Nana");  
console.log(message);
```



# Scope Example

Let's imagine you have a code base of 1 million lines.  
Now, you want to write the following line:

```
var message = "hello";
```

Can you do it?

Well, it depends. Was `message` declared as a variable before?

Now you have to check in the 1 million lines of code...

# Scope

## Global Variables in JS

In JavaScript, the global scope is the complete environment.

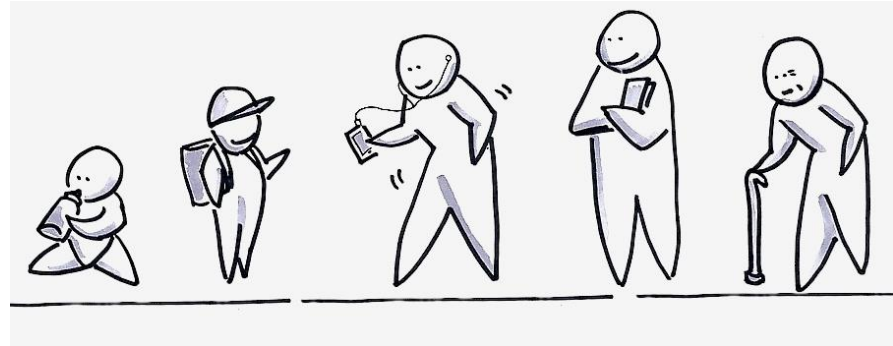
In the browser, the global scope is the window object.  
All global variables belong to the window object.

```
var global_var = 6;
```

```
console.log(window.global_var);
```

# Variable Lifetime

## Variable Lifetime



**Global variables** live as long as your application (your web page) lives.



**Local variables** have short lives. They are created when the function is invoked, and deleted when the function is finished.



# Questions

```
console.log("Questions?");
```

# Namespaces

How can we structure our code to look good, be organized and clear?



## Meet Namespaces

# Naming variables and object namespaces

```
var greet = "Hello";  
var greet = "Hola";  
  
console.log(greet);  
  
// what is going to print?  
// will we lose data?
```

# Naming variables and object namespaces

```
var english = {};  
var spanish = {};  
  
english.greet = "Hello!";  
spanish.greet = "Hola!";  
  
console.log(english.greet);  
console.log(spanish.greet);  
  
// whats will be printed here?
```

# Namespace

## Definition

"A namespace is a container which allows developers to bundle up functionality under a unique, application-specific name.

**In JavaScript a namespace is just another object containing methods, properties, and objects." ---MDN**

```
var myNameSpace = { } ;
```

- ▶ This defines a unique scope.
- ▶ Different from the global scope



# What is a namespace

- ▶ A namespace is a container for variables and functions.
- ▶ Typically to keep variables and functions with the same name separate.
- ▶ And to group variables and functions that are related together.

# Namespace example

**In another program of a game (with 2 players and a board) you see the following lines:**

```
drawCard(card);  
addCard(card2);
```

**What is this code doing? Who does it belong to?**

**But if we had these lines:**

```
Player1.drawCard(card);  
Board.addCard(card2);
```

**Or we could have had these lines:**

```
Board.drawCard(card);  
Player1.addCard(card2);
```

**Context gives us a meaning and clarification.**

# Question

Raise your hand:

**How many people throw their clothes on the floor?**

Some people think this is the best way to organize their room.

If you are that kind of a person maybe you will find it challenging to understand namespaces.

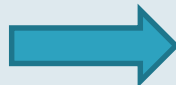


# Namespace example

```
var board = [];  
var player1Score = 30;  
var player2Name = "Mimi";  
  
function updatePlayerScore(playerScore){  
  if (playerScore < 10){  
    return playerScore + 2;  
  } else {  
    return playerScore + 5;  
  }  
}  
  
var player2Score = 20;  
var player1Name = "Momo";  
function printWinner(){  
  var winner = player1Score > player2Score ? player1Name : player2Name;  
  console.log(winner + " won!");  
}  
  
function addCard(card){  
  if (board.length < 15){  
    board.push(card);  
  }  
}
```

# Game refactor

```
var game = {  
  board: [],  
  player1: {  
    score: 30,  
    name: "Momo"  
  },  
  player2: {  
    score: 20,  
    name: "Mimi"  
  },  
  updatePlayerScore: updatePlayerScore,  
  printWinner: printWinner,  
  addCard: addCard  
}  
  
function updatePlayerScore(playerScore){  
  //...  
}  
function printWinner(){  
  //...  
}  
function addCard(card){  
  //...  
}
```



Now we also have context.  
The board is not just a  
board it is the board of  
the game.

```
function addCard(card){  
  if (game.board.length < 15){  
    game.board.push(card);  
  }  
}
```

# Questions

```
console.log("Questions?");
```

# Let's Look at this Code

```
function add(value1, value2, value2){  
    console.log(value3);  
    return value1 + value2;  
}  
console.log(add(1,2,3));
```

What is the output?

```
✖ ▶ Uncaught ReferenceError: value3 is not defined  
    at add (<anonymous>:2:14)  
    at <anonymous>:5:13
```

# Use strict

## Putting order into chaos

It seems that JS is a bit chaotic by nature

We don't have to put semicolons

We don't have to use var

We can use any keyword as a variable name

And more...

## Meet "use strict"





# Strict mode

Introduced in ECMAScript 5 (2009), Strict mode limits the things we can do without getting "punished"

We'll see more error messages in the console and in the debugger  
(Not supported on IE9 and below)

We can define it

1. in the scope of a file (at the very top)
2. inside a specific function

```
1  "use strict";
2  function crea
3      var new_e
4      new_eleme
5      return ne
6  }
```

```
1  function createElemen
2  "use strict";
3      var new_element =
4      new_element.class
5      return new_eleme
6  }
```

# Now, with use strict

```
"use strict";

function add(value1, value2, value2){
    console.log(value3);
    return value1 + value2;
}

console.log(add(1,2,3));
```

What is the output?

✖ Uncaught SyntaxError: Duplicate parameter name not allowed in this context

# Questions

```
console.log("Questions?");
```

# Prevent default

- ▶ Some events in HTML/JS have default behavior.
- ▶ For example, clicking on an `<a>` tag (hyperlink), usually takes you to another page.
- ▶ In some cases we want to avoid the default behavior of an event. How do we do that?

`<body>`

I am not going to do anything when you

`<a href="https://www.itc.tech/">click on me!</a>`

`<script>`

```
document.querySelector("a").addEventListener("click", function (e) {  
    e.preventDefault();  
});
```

`</script>`

`</body>`

# Prevent default

- ▶ Another example, is text selection, usually when we click the mouse and drag it over a text, the text is being selected.
- ▶ In some cases we want to avoid that default behavior

```
<body>
  <h2>This text cannot be selected</h2>
  <script>
    document.querySelector("h2").addEventListener("click", function (e) {
      e.preventDefault();
    });
  </script>
</body>
```

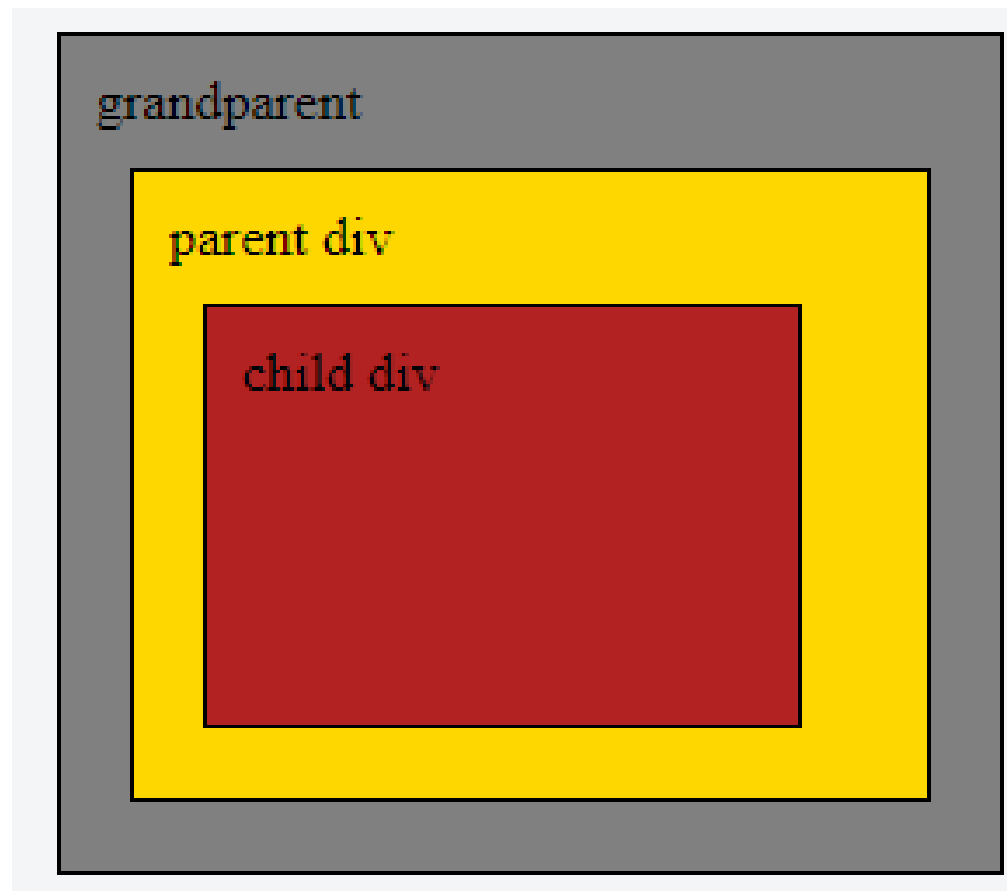
Demo !

# Questions

```
console.log("Questions?");
```

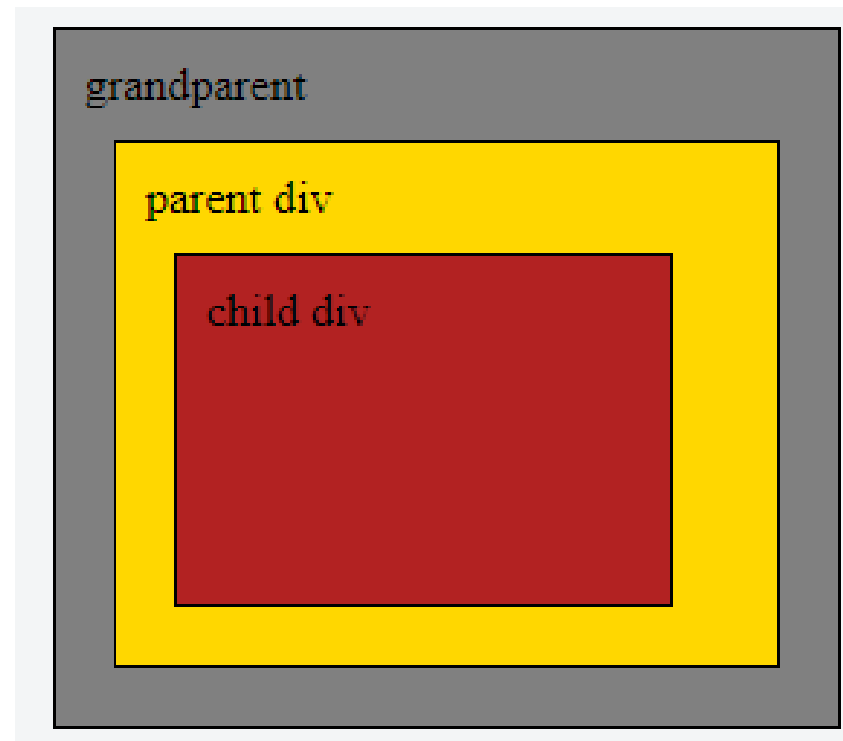
# Event propagation

- ▶ Imagine three nested divs.



# Event propagation

- ▶ We shall call the outer div the "grandparent" div
- ▶ The middle div will be the "parent" div
- ▶ And the inner div the "child" div.

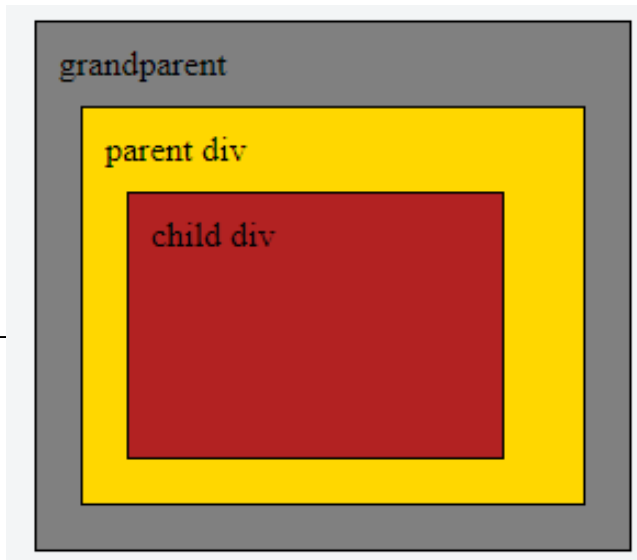




# Event propagation

## HTML:

```
<div id="grandparent">
  grandparent
  <div id="parent">
    parent div
    <div id="child">
      child div
    </div>
  </div>
</div>
```

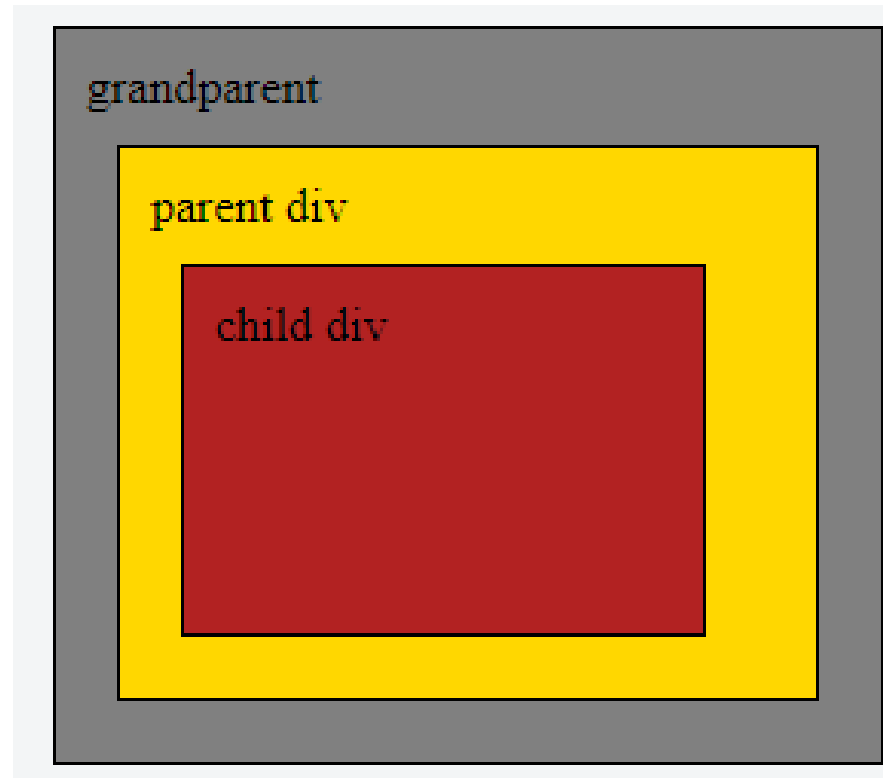


## CSS:

```
div {
  border: solid 1px black;
  padding: 10px;
  margin: 10px;
}
.grandparent {
  background: gray;
  width: 250px;
}
.parent {
  background: gold;
}
.child {
  height: 100px;
  width: 150px;
  background: firebrick;
}
```

# Event propagation

- ▶ Let's see it in action: [Example](#)

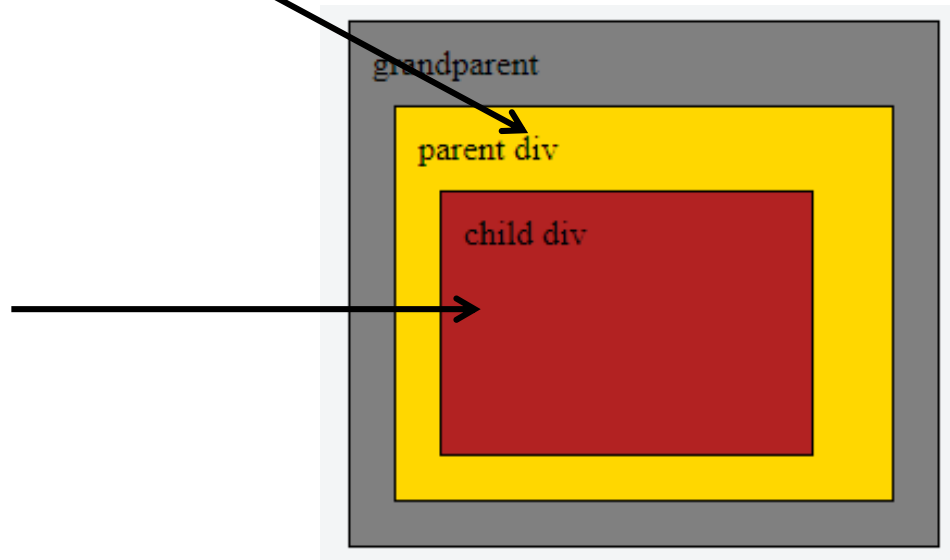


# Event propagation

1. We add an onclick event listener to the parent:

```
document.getElementById("parent").addEventListener("click", function(){  
    console.log("parent clicked");  
});
```

2. We click on the child



What happens when we click the child?

- ▶ The parents event handler gets called!

# Event propagation

- ▶ What's happening?
  - ▶ Why is the parent div handler being called as well?
  - ▶ The reason is called *event propagation*.
  - ▶ The event propagates up from the nested tag, and triggers the parents event handlers as well!
- 
- ▶ Let's see another [example](#)

# Questions

```
console.log("Questions?");
```

# Questions

```
console.log("Questions?");
```

# Closure

**Task:** We want to have a function that will count the times it is being called

Let's try:

```
function count_self_calls(){  
  var counter = 0;  
  counter++;  
  console.log(counter);  
}
```

But it doesn't work!



# Closure

## Second try:

We define a nested function that uses the scope of the outer function:

```
function get_count_self_calls(){  
  var counter = 0;  
  
  return function private(){  
    counter++;  
    console.log(counter);  
  }  
}
```

```
var count_self_calls = get_count_self_calls;  
count_self_calls(); // 1  
count_self_calls(); // 2
```





# Nested Functions

We define a function inside a function

```
function get_count_self_calls(){  
  var counter = 0;  
  
  return function private(){  
    counter++;  
    console.log(counter);  
  }  
}
```

1. The return type of the outer function is a function!

```
var count_self_calls = get_count_self_calls();  
count_self_calls(); // 1  
count_self_calls(); // 2
```

# Nested Functions

We know we can return values from functions:

Return number: 

```
function add1(num){  
  return num + 1;  
}
```

Return string: 

```
function concat(str, str2){  
  return str + str2;  
}
```

Return boolean: 

```
function isEven(num){  
  return num % 2 === 0;  
}
```

**Functions can also return functions:**

```
function getLogHi(){  
  return function(){  
    console.log("hi");  
  }  
}
```

# Nested Functions

```
function getLogHi(){  
  return function logHi(){  
    console.log("hi");  
  }  
}
```

When we invoke the `getLogHi` function we get the `logHi` Function object back.

```
var logHi = getLogHi();
```

`logHi` is a function object. We did not invoke it yet  
`typeof logHi; // "function"`

Now we can invoke `logHi`  
`logHi(); // hi`

When we invoke the function we get a `console.log` of "hi".

# Nested Functions

We can return a function that receives parameters:

```
function getAdd1(){  
  return function (num){  
    return num + 1;  
  }  
}
```

When we invoke the `getAdd1` function we get a function object back.

```
var add1 = getAdd1();  
typeof add1; //function
```

Now we can invoke `add1`

```
add1(3); //?  
//4  
add1(100);//?  
//101
```

# Nested Functions – Functions Factory

We can pass parameters to the outer functions:

```
function getPowerOf(power){  
  return function (number){  
    console.log(Math.pow(number, power));  
    return Math.pow(number, power);  
  }  
}
```

When we invoke the `getPowerOf` function we pass a parameter that sets the power variable.

```
var powerOf2 = getPowerOf(2);
```

Now we can invoke `powerOf2` for all kinds of numbers:

```
powerOf2(2); //?
```

```
//4
```

```
powerOf2(3); //?
```

We can also create `powerOf3` function by calling `getPowerOf` function with 3.

```
var powerOf3 = getPowerOf(3);
```

```
powerOf3(2); //8
```

```
powerOf3(3); //?
```

# Back to the First Example

We define a function inside a function

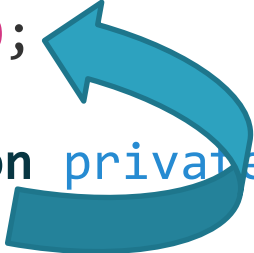
```
function get_count_self_calls(){  
  var counter = 0;  
  
  return function private(){  
    counter++;  
    console.log(counter);  
  }  
}
```

1. The return type of the outer function is a function!

```
var count_self_calls = get_count_self_calls();  
count_self_calls(); // 1  
count_self_calls(); // 2
```

# Nested Functions

```
function get_count_self_calls(){  
  var counter = 0;  
  
  return function private(){  
    counter++;  
    console.log(counter);  
  }  
}
```



2. The inner function can access the scope of the outer function

```
var count_self_calls = get_count_self_calls();  
count_self_calls(); // 1  
count_self_calls(); // 2
```

# Nested Functions

```
function get_count_self_calls(){  
  var counter = 0;  
  
  return function private(){  
    counter++;  
    console.log(counter);  
  }  
}
```

```
var count_self_calls = get_count_self_calls();  
count_self_calls(); // 1  
count_self_calls(); // 2
```

3. When invoking the outer function we get a function!

4. The counter still exist after the outer function is invoked.



# Closures – Explained by Analogue

Whenever you declare a new function and assign it to a variable, you store the function definition, *as well as a **closure***.

The **closure** is like a backpack with all of the variables that were in scope at the time that the function definition was created.



# Questions

```
console.log("Questions?");
```

# Advanced Cheat Sheet

## setInterval

```
var alertInterval = setInterval(function () {  
    alert("This is annoying");  
}, 5000);  
clearInterval(alertInterval);
```

## setTimeout

```
var alertTimeout = setTimeout(my_alert, 5000);  
clearTimeout(alertTimeout);
```

## Pointer Events

```
a {  
    pointer-events: none;  
}
```

## Prevent Default:

```
var link = document.querySelector("a");  
link.addEventListener("click", function (e) {  
    e.preventDefault();  
});
```

## scope

```
var global_var = 6;  
function local_scope(){  
    var local_var = "local";  
    console.log(global_var); //6  
}  
console.log(local_var); //undefined
```

## Nested Functions

```
function get_countCalls() {  
    var counter = 0;  
    return function private() {  
        return counter++;  
    }  
}  
var countCalls = get_countCalls();  
countCalls(); // 1
```