JavaScript Advanced Concepts - Complete Guide

Table of Contents

1. [The `this` Keyword - Five Binding Rules]
2. [Hard Binding]
3. [The `that` Pattern]
4. [Call, Apply, and Bind Methods]
5. [Closures]
6. [Factory Pattern]
7. [Function Constructor Pattern]
8. [Module Context and Strict Mode]

---

1. The `this` Keyword - Five Binding Rules

The `this` keyword in JavaScript is one of the most misunderstood concepts. Its value is determined by **how a function is called**, not where it's defined. There are five main binding rules that determine what `this` refers to.

1.1 Default Binding (Window Object)

Concept: When a function is called in the global scope without any context, `this` refers to the global object (in browsers, this is the `window` object). In strict mode, `this` will be `undefined` instead.

How it works:

- When you declare variables with `var` in the global scope, they become properties of the `window` object
- When you call a function directly (not as a method of an object), the caller is considered to be `window`
- Therefore, `this` inside that function refers to `window`

Code Example Explanation:

```
var fname = "mona";

function getName() {
```

```
    console.log(this.fname); // 'this' refers to window
}

getName(); // Output: "mona"
// This is equivalent to: window.getName()
```

Deep Dive:

- JavaScript's execution context has two phases: creation phase and execution phase
- During the creation phase, the JavaScript engine creates the global execution context
- In this context, `this` is bound to the global object by default
- When `getName()` is invoked without a dot notation or explicit binding, the default binding rule applies
- `this.fname` becomes `window.fname`, which equals "mona"

Important Notes:

- This only works with `var`, not `let` or `const` (which don't attach to the window object)
- In strict mode (`'use strict'`), default binding sets `this` to `undefined` instead of `window`
- This is the most common source of bugs when developers expect `this` to be something else

1.2 Implicit Binding (Object Context)

Concept: When a function is called as a method of an object (using dot notation), `this` refers to the object that owns the method - the object to the left of the dot.

How it works:

- The call-site (where the function is invoked) determines the binding
- If there's an object reference at the call-site, that object becomes `this`
- The object that "owns" or "contains" the function at call-time becomes the context

Code Example Explanation:

```
var person = {
    fname: "Ahmed",
    password: "SMFKF",
    getPassword: function() {
        console.log(this.password);
    }
};
```

```
person.getPassword(); // Output: "SMFKF"
// 'this' refers to 'person' because person is the caller
```

Deep Dive:

- When `person.getPassword()` is executed, JavaScript looks at what's to the left of the dot
- That object (`person`) becomes the binding for `this`
- Inside `getPassword` , `this.password` resolves to `person.password`
- This binding is called "implicit" because it happens automatically based on the call-site

The Lost Binding Problem:

```
var fname = "mona";
var password = "PD";

var person = {
    fname: "Ahmed",
    password: "SMFKF",
    getPassword: function() {
        console.log(this.password);
    }
};

let result = person.getPassword; // Assigning function reference
result(); // Output: "PD" (not "SMFKF"!)
```

Why does this happen?

- When you assign `person.getPassword` to `result` , you're only copying the function reference
- You're not binding the context
- When `result()` is called, there's no object to the left of the call
- Default binding takes over, so `this` becomes `window`
- `this.password` resolves to `window.password` which is "PD"

Key Takeaway: Implicit binding is lost when you pass methods as callbacks or assign them to variables without preserving the context.

1.3 Explicit Binding (Call, Apply, Bind)

Concept: You can explicitly specify what `this` should be using the methods `call()` , `apply()` , or `bind()` . This overrides both default and implicit binding.

How it works:

- These are methods available on every function object
- They allow you to manually set the `this` value
- You're telling JavaScript exactly what context to use

Call Method:

```
var fname = "mona";

function getName() {
    console.log(this.fname);
}

var person = {
    fname: "ahmed",
    display: function() {
        getName.call(this); // Explicitly setting 'this' to person
    }
};

person.display(); // Output: "ahmed"
```

Deep Dive on Call:

- `getName.call(this)` means: "Execute getName, but use this specific object as `this`"
- Inside `person.display()`, `this` refers to `person` (implicit binding)
- When we do `getName.call(this)`, we're passing `person` as the context
- So inside `getName`, `this.fname` becomes `person.fname` = "ahmed"
- Without `.call(this)`, `getName()` would use default binding and access `window.fname`

Syntax:

- `functionName.call(thisArg, arg1, arg2, ...)`
- First parameter: the object to use as `this`
- Remaining parameters: arguments to pass to the function

Apply Method: Same as `call`, but arguments are passed as an array:

```
person1.display.apply(person2, [":)", ":("]);
// vs
person1.display.call(person2, ":)", ":(");
```

Bind Method: Creates a new function with `this` permanently bound:

```
var output = person1.display.bind(person2);
output(); // 'this' is always person2, regardless of how it's called
```

Key Difference:

- `call` and `apply` : Execute the function immediately
- `bind` : Returns a new function with `this` bound permanently (hard binding)

1.4 New Binding (Constructor Functions)

Concept: When a function is called with the `new` keyword, a brand new object is created, and `this` refers to that new object.

How it works - What `new` does (4 steps):

1. Creates a brand new empty object
2. Links this object to the function's prototype
3. Sets `this` to point to this new object
4. Returns this object (unless the function explicitly returns a different object)

Code Example Explanation:

```
function Product(name, price, category) {
    this.name = name;
    this.price = price;
    this.category = category;
}

Product.prototype.discount = function() {
    console.log("discount function");
};

var p1 = new Product("Book", 200, "school");
```

Step-by-Step Execution:

1. `new Product(...)` is called
2. JavaScript creates: `var this = {};` (a new empty object)
3. JavaScript sets up prototype chain: `this.__proto__ = Product.prototype`
4. The constructor runs:
    - `this.name = "Book"` (adds property to the new object)

- `this.price = 200`
- `this.category = "school"`

5. JavaScript implicitly returns `this`
6. `p1` now holds the reference to this new object

Deep Dive:

```
var p1 = new Product("Book", 200, "school");
var p2 = new Product("phone", 3000, "electronics");

console.log(p1);
// Output: Product { name: "Book", price: 200, category: "school" }

console.log(p2);
// Output: Product { name: "phone", price: 3000, category: "electronics" }
```

Why this is powerful:

- Each call to `new Product()` creates a completely new object
- Each object has its own properties (name, price, category)
- But they all share the same methods via the prototype (memory efficient)
- `p1.discount === p2.discount` returns `true` (same function reference)

Prototype Chain Explanation:

- Methods added to `Product.prototype` are shared by all instances
- When you call `p1.discount()`, JavaScript:
  1. Looks for `discount` on `p1` directly - doesn't find it
  2. Looks up the prototype chain to `Product.prototype` - finds it there
  3. Executes it with `this` bound to `p1`

Without `new` keyword:

```
var p3 = Product("Pen", 50, "office"); // Forgot 'new'
console.log(p3); // undefined
console.log(window.name); // "Pen" - Oops! Added to global object
```

This is why constructor functions are conventionally capitalized - to remind you to use `new`.

1.5 Lexical Binding (Arrow Functions)

Concept: Arrow functions don't have their own `this` binding. They inherit `this` from the enclosing (parent) lexical scope at the time they are defined, not when they are called.

How it works:

- Arrow functions capture the `this` value from their surrounding context
- This binding is permanent and cannot be changed with `call`, `apply`, or `bind`
- "Lexical" means it's determined by where the function is written in the code, not how it's called

The Problem Arrow Functions Solve:

```
var fname = "mona";

var person = {
    fname: "Ahmed",
    display: function() {
        setTimeout(function() {
            console.log(this.fname); // Output: "mona" (not "Ahmed"!)
        }, 1000);
    }
};

person.display();
```

Why does this fail?

- `person.display()` is called, so inside `display`, `this` = `person` ✓
- But `setTimeout` executes the callback function later
- When the callback runs, it's not called as a method of `person`
- It's called by the setTimeout mechanism, which uses default binding
- So `this` inside the callback becomes `window`
- Result: `this.fname` = `window.fname` = "mona"

Solution with Arrow Function:

```
var fname = "mona";

var person = {
    fname: "Ahmed",
    display: function() {
        setTimeout(() => {
            console.log(this.fname); // Output: "Ahmed" ✓
```

```
        }, 1000);
    }
};

person.display();
```

Why does this work?

- The arrow function doesn't create its own `this`
- It looks up to the enclosing scope (the `display` function)
- In `display`, `this` = `person` (implicit binding)
- The arrow function "captures" this `this` value
- Even when the arrow function executes later, it still uses the captured `person` context
- Result: `this.fname` = `person.fname` = "Ahmed"

Deep Dive on Lexical Scope:

```
var person = {
    fname: "Ahmed",
    display: function() {
        console.log(this); // person object

        setTimeout(() => {
            console.log(this.fname); // "Ahmed"
        }, 1000);
    }
};
```

Think of it as the arrow function saying: "I don't care who calls me or how - I'm using the `this` from where I was born (defined)."

Important Limitations:

```
var obj = {
    fname: "Test",
    getName: () => {
        console.log(this.fname); // Won't work as expected!
    }
};

obj.getName(); // Output: undefined (not "Test")
```

Why?

- The arrow function is defined in the global scope
- Its lexical parent is the global scope, not the object
- So `this` refers to `window`, not `obj`
- Lesson: Don't use arrow functions as object methods if you need `this` to refer to the object

When to use arrow functions:

- Callbacks (setTimeout, setInterval, event handlers)
- Array methods (map, filter, forEach) where you want to preserve outer `this`
- Any situation where you want `this` from the enclosing scope

When NOT to use arrow functions:

- Object methods that need `this` to refer to the object
- Constructor functions (arrow functions can't be used with `new`)
- Methods that need their own `this` context

---

2. Hard Binding

Concept: Hard binding is a technique to permanently fix the `this` context of a function so it cannot be overridden, even by explicit binding methods like `call` or `apply`.

The Problem It Solves: Normal binding can be lost or overridden:

```
var person1 = {
    fname: "ahmed",
    display: function() {
        console.log(this.fname);
    }
};

var callback = person1.display;
callback(); // Lost binding - 'this' is now window
```

Solution - Using `.bind()`:

```
var person1 = {
    fname: "ahmed",
    lname: "alaa",
    display: function(emoji1, emoji2) {
        console.log(emoji1 + " " + this.fname + " " + this.lname + " " +
```

```
emoji2);
    }
};

var person2 = {
    fname: "Sama",
    lname: "Mohamed"
};

var output = person1.display.bind(person2);
output(":)", ":(");
// Output: ":) Sama Mohamed :("
```

Deep Dive - What `.bind()` Does:

1. Creates a completely new function
2. This new function has `this` permanently set to whatever object you passed to `bind`
3. Returns this new function (doesn't execute it immediately)
4. The binding is "hard" - it cannot be changed later

The Hard Binding is Permanent:

```
var output = person1.display.bind(person2);

// Even if you try to override with call:
output.call(person1, ":)", ":(");
// Still outputs: ":) Sama Mohamed :("
// The binding to person2 cannot be changed!
```

How This Works Internally: When you call `.bind()`, JavaScript essentially does this:

```
// Simplified version of what bind does
Function.prototype.bind = function(context) {
    var fn = this; // The original function
    return function() {
        return fn.apply(context, arguments);
    };
};
```

So `output` is a new function that internally always calls the original with a fixed context.

Practical Use Case - Event Handlers:

```
var counter = {
    count: 0,
    increment: function() {
        this.count++;
        console.log(this.count);
    }
};

// Problem:
button.addEventListener('click', counter.increment);
// 'this' inside increment will be the button, not counter!

// Solution:
button.addEventListener('click', counter.increment.bind(counter));
// Now 'this' is always counter
```

Partial Application with Bind: You can also pre-set arguments:

```
function multiply(a, b) {
    return a * b;
}

var double = multiply.bind(null, 2); // Pre-set first argument to 2
console.log(double(5)); // Output: 10 (2 * 5)
console.log(double(10)); // Output: 20 (2 * 10)
```

Key Differences:

- **Soft Binding** (`call` / `apply`): Executes immediately, binding can be overridden
- **Hard Binding** (`bind`): Returns new function, binding is permanent

Memory Consideration: Each call to `.bind()` creates a new function object:

```
var fn1 = person1.display.bind(person2);
var fn2 = person1.display.bind(person2);

console.log(fn1 === fn2); // false - different function objects
```

This can matter in situations where you need to remove event listeners:

```
// Won't work - different function reference
element.addEventListener('click', fn.bind(this));
element.removeEventListener('click', fn.bind(this));
```

```
// Correct way
var boundFn = fn.bind(this);
element.addEventListener('click', boundFn);
element.removeEventListener('click', boundFn);
```

---

3. The `that` Pattern

Concept: The `that` pattern (also called `self` pattern) is an old-school technique used before arrow functions existed to preserve the `this` context in nested functions or callbacks.

The Problem:

```
var fname = "mona";

var person = {
    fname: "Ahmed",
    display: function() {
        setTimeout(function() {
            console.log(this.fname); // "mona" - wrong!
        }, 1000);
    }
};

person.display();
```

Inside the setTimeout callback, `this` refers to `window`, not `person`.

The `that` Pattern Solution:

```
var fname = "mona";

var person = {
    fname: "Ahmed",
    display: function() {
        var that = this; // Save reference to the correct 'this'

        setTimeout(function() {
            console.log(that.fname); // "Ahmed" - correct!
        }, 1000);
    }
};
```

```
person.display();
```

How It Works - Deep Dive:

1. **Closure Mechanism**:
   - When `display()` is called, `this` refers to `person` (implicit binding)
   - We create a variable `var that = this;` which saves this reference
   - `that` is now a regular variable containing a reference to the `person` object
2. **Closure Preservation**:
   - The setTimeout callback function forms a closure
   - Closures capture variables from their outer scope
   - `that` is in the outer scope, so the callback has access to it
   - Even when the callback executes later, it still has access to `that`
3. **Why This Works**:
   - `that` is not affected by how the function is called
   - It's just a normal variable holding an object reference
   - The closure keeps this reference alive
   - Unlike `this`, `that` doesn't change based on call-site

Step-by-Step Execution:

```
// Step 1: person.display() is called
var person = {
    fname: "Ahmed",
    display: function() {
        // Step 2: At this point, 'this' = person
        var that = this; // that now holds reference to person object

        // Step 3: setTimeout is set up
        setTimeout(function() {
            // Step 4 (1 second later): Callback executes
            // 'this' would be window here
            // But 'that' still refers to person (via closure)
            console.log(that.fname); // Accesses person.fname
        }, 1000);
    }
};
```

Naming Conventions: Different codebases use different names for the same pattern:

```
var that = this;  // Most common
var self = this;  // Also popular
var me = this;    // Sometimes used
var _this = this; // Occasionally seen
```

Multiple Levels of Nesting:

```
var obj = {
    name: "Object",
    method1: function() {
        var that = this; // Saves obj reference

        setTimeout(function() {
            console.log(that.name); // "Object"

            setTimeout(function() {
                console.log(that.name); // Still "Object"
                // 'that' is accessible due to closure chain
            }, 1000);
        }, 1000);
    }
};
```

Common Pitfall - Confusion with Multiple `that`s:

```
var outer = {
    name: "Outer",
    inner: {
        name: "Inner",
        method: function() {
            var that = this; // Refers to 'inner'

            setTimeout(function() {
                console.log(that.name); // "Inner"
            }, 1000);
        }
    }
};
```

Modern Alternative - Arrow Functions: Today, arrow functions are the preferred solution:

```
var person = {
    fname: "Ahmed",
```

```
    display: function() {
        setTimeout(() => {
            console.log(this.fname); // "Ahmed" - no need for 'that'
        }, 1000);
    }
};
```

Why Arrow Functions Replaced `that` Pattern:

1. **Cleaner syntax**: No need for extra variable
2. **Less confusion**: Fewer variables to track
3. **Lexical this**: More intuitive behavior
4. **Less error-prone**: Can't accidentally use wrong variable name

When You Might Still See `that` Pattern:

1. Legacy codebases (pre-ES6)
2. Environments that don't support arrow functions
3. Code that needs to support old browsers
4. Personal preference of developers familiar with the pattern

Performance Note: Both patterns have similar performance - the arrow function approach is just more modern and readable.

---

4. Call, Apply, and Bind Methods

These three methods allow you to explicitly control what `this` refers to in a function. They're fundamental to understanding JavaScript's execution context.

4.1 The `call()` Method

Syntax: `functionName.call(thisArg, arg1, arg2, arg3, ...)`

Purpose: Immediately invokes the function with a specified `this` value and individual arguments.

Deep Explanation:

```
var person1 = {
    fname: "ahmed",
    lname: "alaa",
    display: function(emoji1, emoji2) {
```

```
        console.log(emoji1 + " " + this.fname + " " + this.lname + " " +
emoji2);
    }
};

var person2 = {
    fname: "Sama",
    lname: "Mohamed"
};

person1.display.call(person2, ":)", ":(");
// Output: ":) Sama Mohamed :("
```

What Happens:

1. We're calling the `display` method that belongs to `person1`
2. But we're telling it to use `person2` as `this`
3. The arguments `:)` and `:(` are passed as normal parameters
4. Inside `display`, `this.fname` resolves to `person2.fname` = "Sama"
5. The function executes immediately

Use Cases:

- Borrowing methods from other objects
- Calling parent constructors in inheritance
- Explicitly setting context for callbacks

Method Borrowing Example:

```
var numbers = {
    data: [1, 2, 3, 4, 5],
    sum: function() {
        return this.data.reduce((a, b) => a + b, 0);
    }
};

var moreNumbers = {
    data: [10, 20, 30]
};

var result = numbers.sum.call(moreNumbers);
console.log(result); // 60 - borrowed sum method for moreNumbers
```

Constructor Chaining:

```
function Person(name) {
    this.name = name;
}

function Employee(name, title) {
    Person.call(this, name); // Call parent constructor
    this.title = title;
}

var emp = new Employee("John", "Developer");
// emp.name = "John", emp.title = "Developer"
```

4.2 The `apply()` Method

Syntax: `functionName.apply(thisArg, [argsArray])`

Purpose: Same as `call()`, but arguments are passed as an array.

Deep Explanation:

```
person1.display.apply(person2, ["^_^", ":=)"]);
// Output: "^_^ Sama Mohamed :=)"

// This is functionally equivalent to:
person1.display.call(person2, "^_^", ":=)");
```

The Key Difference:

- `call`: Arguments passed individually
- `apply`: Arguments passed as an array

Why This Matters - Practical Example:

```
var numbers = [5, 6, 2, 3, 7];

// Using apply with Math.max
var max = Math.max.apply(null, numbers);
console.log(max); // 7

// Without apply, you'd need:
var max = Math.max(5, 6, 2, 3, 7);
```

How This Works:

- `Math.max()` expects individual arguments: `Math.max(5, 6, 2, 3, 7)`
- We have an array: `[5, 6, 2, 3, 7]`
- `apply` "spreads" the array into individual arguments
- It's like doing: `Math.max.call(null, 5, 6, 2, 3, 7)`

Modern Alternative - Spread Operator:

```
var numbers = [5, 6, 2, 3, 7];
var max = Math.max( ... numbers); // ES6 spread operator
```

Another Use Case - Array Concatenation:

```
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

Array.prototype.push.apply(arr1, arr2);
console.log(arr1); // [1, 2, 3, 4, 5, 6]

// Modern way:
arr1.push( ... arr2);
```

Dynamic Arguments Example:

```
function greet(greeting, punctuation) {
    console.log(greeting + ", " + this.name + punctuation);
}

var person = { name: "Alice" };
var args = ["Hello", "!"];

greet.apply(person, args); // "Hello, Alice!"
```

4.3 The `bind()` Method

Syntax: `var newFunc = functionName.bind(thisArg, arg1, arg2, ...)`

Purpose: Creates a NEW function with a permanently bound `this` value. Does NOT execute immediately.

Deep Explanation:

```
var person1 = {
    fname: "ahmed",
```

```
    lname: "alaa",
    display: function(emoji1, emoji2) {
        console.log(emoji1 + " " + this.fname + " " + this.lname + " " +
emoji2);
    }
};

var person2 = {
    fname: "Sama",
    lname: "Mohamed"
};

// bind returns a NEW function
var output = person1.display.bind(person2);

// Call it later
output(":)", ":("); // ":) Sama Mohamed :("
```

Critical Understanding:

1. `bind()` does NOT execute the function
2. It returns a brand new function
3. This new function has `this` locked to what you passed to `bind`
4. You can call this new function whenever you want

The Permanent Binding:

```
var output = person1.display.bind(person2);

// Try to override with call - won't work
output.call(person1, ":)", ":(");
// Still outputs: ":) Sama Mohamed :("
// The binding to person2 is permanent!
```

Partial Application (Currying):

```
function multiply(a, b, c) {
    return a * b * c;
}

// Bind first argument
var double = multiply.bind(null, 2);
console.log(double(3, 4)); // 2 * 3 * 4 = 24
```

```
// Bind first two arguments
var doubleAndTriple = multiply.bind(null, 2, 3);
console.log(doubleAndTriple(4)); // 2 * 3 * 4 = 24
```

Event Handler Use Case:

```
var button = {
    clicks: 0,
    handleClick: function() {
        this.clicks++;
        console.log("Clicked " + this.clicks + " times");
    }
};

// Problem:
document.getElementById("myButton")
    .addEventListener("click", button.handleClick);
// 'this' will be the DOM element, not button object

// Solution:
document.getElementById("myButton")
    .addEventListener("click", button.handleClick.bind(button));
// Now 'this' is always the button object
```

React Class Components Classic Example:

```
class MyComponent {
    constructor() {
        this.state = { count: 0 };
        // Bind in constructor to avoid creating new function each render
        this.handleClick = this.handleClick.bind(this);
    }

    handleClick() {
        this.setState({ count: this.state.count + 1 });
    }

    render() {
        return <button onClick={this.handleClick}>Click</button>;
    }
}
```

4.4 Comparison Table

Feature	call	apply	bind
Executes immediately	✓	✓	✗ (returns new function)
Arguments format	Individual	Array	Individual
Returns	Function result	Function result	New function
Binding permanence	One-time	One-time	Permanent
Use case	Quick method borrowing	Array arguments	Event handlers, callbacks

Performance Notes:

- `call` is slightly faster than `apply` (no array processing)
- `bind` creates a new function (slight memory overhead)
- In modern JS, spread operator `...` often replaces `apply`

Common Interview Question: "What's the difference between call, apply, and bind?"

Answer:

- All three control what `this` refers to
- `call` and `apply` execute immediately; `bind` returns a new function
- `call` takes arguments individually; `apply` takes them as an array
- `bind` creates permanent binding that can't be overridden

---

5. Closures

Concept: A closure is a function that has access to variables from its outer (enclosing) function's scope, even after that outer function has finished executing. This is one of JavaScript's most powerful and unique features.

5.1 Basic Closure

Simple Example:

```
function showData() {
    var trackName = "pd";

    return function() {
        console.log("hello " + trackName);
```

```
    };
}

var result = showData();
result(); // Output: "hello pd"
```

What's Happening - Deep Dive:

1. **Execution Phase**:
   - `showData()` is called
   - A new execution context is created
   - Variable `trackName` is created in this context with value "pd"
   - An anonymous function is created (the one being returned)
   - This function is returned and assigned to `result`
   - `showData()` finishes executing
2. **The Mystery**:
   - Normally, when a function finishes, its execution context is destroyed
   - All its local variables would be garbage collected
   - But here, `trackName` should be gone, right?
   - Wrong! This is where closures come in
3. **Closure Magic**:
   - The inner function (returned function) maintains a reference to its outer scope
   - Even though `showData` has finished, `trackName` is kept alive
   - When `result()` is called later, it still has access to `trackName`
   - This is because the inner function "closes over" the outer scope

Memory Visualization:

```
[Global Scope]
  - result: [Function]
    |
    └──> [Closure Scope] (preserved)
           - trackName: "pd"
           - function code that references trackName
```

5.2 Counter Example - Closures with State

Code:

```
function counter() {
    var count = 0;
```

```
    return function() {
        console.log(count++);
    };
}

var result = counter();
result(); // 0
result(); // 1
result(); // 2
result(); // 3
```

Deep Analysis:

1. **First Call** `counter()`:
   - Creates execution context
   - Initializes `count = 0`
   - Returns inner function
   - Inner function has closure over `count`
2. **First** `result()`:
   - Accesses `count` through closure (value is 0)
   - Logs 0
   - Increments `count` to 1 (post-increment `count++`)
   - **Important**: `count` persists in memory
3. **Second** `result()`:
   - Accesses same `count` (now 1)
   - Logs 1
   - Increments to 2
4. **Key Insight**:
   - Each call to `counter()` creates a NEW closure with its own `count`
   - But calls to `result()` share the SAME closure

Multiple Independent Counters:

```
var counter1 = counter();
counter1(); // 0
counter1(); // 1

var counter2 = counter();
counter2(); // 0 (separate closure!)
counter2(); // 1
```

```
counter1(); // 2 (independent from counter2)
```

Why This Works:

- Each call to `counter()` creates a new execution context
- Each context has its own `count` variable
- Each returned function closes over its own `count`
- They don't interfere with each other

5.3 Data Privacy with Closures

Concept: Closures enable private variables - data that can't be accessed directly from outside.

Example:

```
function createUser(name, role) {
    var lastLogin = null; // Private variable

    return {
        login: function() {
            lastLogin = new Date();
            console.log(name + " logged as " + role);
        },
        getLastLogin: function() {
            return lastLogin;
        }
    };
}

var user1 = createUser("sara", "admin");
user1.login(); // "sara logged as admin"
console.log(user1.getLastLogin()); // [Current Date]
console.log(user1.lastLogin); // undefined - can't access directly!
```

Why This is Powerful:

1. `lastLogin` is completely private
2. Can only be accessed through `getLastLogin()` method
3. Can only be modified through `login()` method
4. You control the interface to your data

Attempting Direct Access:

```
console.log(user1.name); // undefined
console.log(user1.role); // undefined
console.log(user1.lastLogin); // undefined

// The only way to interact is through the methods
user1.login();
user1.getLastLogin();
```

Module Pattern - Advanced Use:

```
var bankAccount = (function() {
    var balance = 0; // Private

    return {
        deposit: function(amount) {
            if (amount > 0) {
                balance += amount;
                return "Deposited " + amount;
            }
        },
        withdraw: function(amount) {
            if (amount <= balance) {
                balance -= amount;
                return "Withdrew " + amount;
            }
            return "Insufficient funds";
        },
        getBalance: function() {
            return balance;
        }
    };
})();

bankAccount.deposit(100); // "Deposited 100"
bankAccount.withdraw(30); // "Withdrew 30"
console.log(bankAccount.getBalance()); // 70
console.log(bankAccount.balance); // undefined - private!
```

5.4 The Classic Loop Problem

The Problem:

```
function getArr() {
    var arr = [];
```

```
    for (var i = 0; i < 3; i++) {
        arr.push(function() {
            console.log(i);
        });
    }

    return arr;
}

let result = getArr();
result[0](); // 3 (not 0!)
result[1](); // 3 (not 1!)
result[2](); // 3 (not 2!)
```

Why Does This Happen?

1. **Loop Execution**:
   - Loop runs 3 times (i = 0, 1, 2)
   - Each iteration pushes a function to the array
   - All three functions close over the SAME `i` variable
   - Loop finishes with `i = 3`
2. **Function Execution**:
   - When we call `result[0]()`, it looks up `i`
   - The `i` it finds is 3 (the final value after the loop)
   - All three functions share this same `i`

Visualization:

```
[arr[0] function] ──┐
[arr[1] function] ──┼──> [Closure: i = 3]
[arr[2] function] ──┘
```

Solution 1 - IIFE (Immediately Invoked Function Expression):

```
function getArr() {
    var arr = [];

    for (var i = 0; i < 3; i++) {
        (function(j) { // IIFE with parameter
            arr.push(function() {
                console.log(j);
            });
```

```
        })(i); // Pass current i as argument
    }

    return arr;
}

let result = getArr();
result[0](); // 0 ✓
result[1](); // 1 ✓
result[2](); // 2 ✓
```

Why This Works:

- Each IIFE creates a NEW scope
- Each scope has its own `j` parameter
- `j` gets the value of `i` at that iteration
- Each pushed function closes over a different `j`

Solution 2 - `let` (ES6):

```
function getArr() {
    var arr = [];

    for (let i = 0; i < 3; i++) { // Changed var to let
        arr.push(function() {
            console.log(i);
        });
    }

    return arr;
}

let result = getArr();
result[0](); // 0 ✓
result[1](); // 1 ✓
result[2](); // 2 ✓
```

Why This Works:

- `let` is block-scoped
- Each iteration of the loop creates a NEW `i`
- Each function closes over its own iteration's `i`

Solution 3 - Create Closure Function:

```
function getArr() {
    var arr = [];

    function makeLogger(value) {
        return function() {
            console.log(value);
        };
    }

    for (var i = 0; i < 3; i++) {
        arr.push(makeLogger(i));
    }

    return arr;
}
```

Why This Works:

- `makeLogger` creates a new closure for each call
- Each closure has its own `value` parameter
- Each returned function closes over a different `value`

5.5 Closures in Real-World Applications

1. Event Handlers:

```
function setupButtons() {
    var buttons = document.querySelectorAll('.btn');

    for (let i = 0; i < buttons.length; i++) {
        buttons[i].addEventListener('click', function() {
            console.log("Button " + i + " clicked");
        });
    }
}
```

2. Debouncing:

```
function debounce(func, delay) {
    let timeoutId;

    return function() {
        var context = this;
        var args = arguments;
```

```
        clearTimeout(timeoutId);
        timeoutId = setTimeout(function() {
            func.apply(context, args);
        }, delay);
    };
}

var debouncedSearch = debounce(function(query) {
    console.log("Searching for: " + query);
}, 300);
```

3. Memoization (Caching):

```
function memoize(fn) {
    var cache = {};

    return function(arg) {
        if (cache[arg]) {
            console.log("From cache");
            return cache[arg];
        }

        console.log("Calculating");
        var result = fn(arg);
        cache[arg] = result;
        return result;
    };
}

var factorial = memoize(function(n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
});

factorial(5); // Calculating... 120
factorial(5); // From cache... 120
```

Memory Considerations:

- Closures keep variables in memory
- Too many closures can lead to memory leaks
- Be mindful when creating many closures with large data

Common Mistake:

```
// Don't do this in a loop with many iterations
for (var i = 0; i < 1000000; i++) {
    setTimeout(function() {
        console.log(i); // Creates 1 million closures!
    }, i);
}
```

Benefits of Closures:

1. Data privacy/encapsulation
2. Function factories
3. Maintaining state
4. Callbacks and event handlers
5. Module pattern
6. Partial application and currying

---

6. Factory Pattern

Concept: The Factory Pattern is a design pattern that uses functions to create and return objects without using the `new` keyword or constructor functions. It's a way to create multiple similar objects with shared structure but different data.

6.1 Basic Factory Pattern

Simple Implementation:

```
function product(_id, _name, _price) {
    return {
        id: _id || "",
        name: _name || "",
        price: _price || "",
        discount: function(rate) {
            return this.price * rate;
        }
    };
}

var p1 = product(1, "Book", 200);
var p2 = product(2, "Pen", 400);

console.log(p1);
```

```
// { id: 1, name: "Book", price: 200, discount: [Function] }

console.log(p2);
// { id: 2, name: "Pen", price: 400, discount: [Function] }
```

How It Works:

1. `product` is a regular function (not a constructor)
2. It returns a new object literal each time it's called
3. No `new` keyword needed
4. Each object gets its own copy of all properties and methods

Deep Dive - What Happens:

```
var p1 = product(1, "Book", 200);

// Internally:
// 1. product function is called
// 2. A new object literal is created
// 3. Properties are assigned:
//    - id: 1
//    - name: "Book"
//    - price: 200
//    - discount: function...
// 4. This object is returned
// 5. p1 now references this object
```

Using the Objects:

```
console.log(p1.name); // "Book"
console.log(p1.discount(0.1)); // 20 (10% of 200)
console.log(p2.discount(0.2)); // 80 (20% of 400)
```

6.2 Factory Pattern with Private Data (Closures)

Advanced Implementation:

```
function createUser(name, role) {
    var lastLogin = null; // Private variable

    return {
        login: function() {
            lastLogin = new Date();
```

```
            console.log(name + " logged as " + role);
        },
        getLastLogin: function() {
            return lastLogin;
        }
    };
}

var user1 = createUser("sara", "admin");
user1.login(); // "sara logged as admin"
console.log(user1.getLastLogin()); // Current date/time

var user2 = createUser("ahmed", "guest");
user2.login(); // "ahmed logged as guest"
```

Why This is Powerful:

1. **Data Encapsulation**:
   - `name`, `role`, and `lastLogin` are private
   - Can't be accessed directly: `user1.name` is `undefined`
   - Can only be used through the methods
2. **Closure Magic**:
   - The returned methods close over the private variables
   - Each user object has its own private data
   - `user1` and `user2` have separate `lastLogin` values

Visualization:

```
user1 ──> { login: [Function], getLastLogin: [Function] }
              │
              └──> [Closure Scope]
                    - name: "sara"
                    - role: "admin"
                    - lastLogin: Date object

user2 ──> { login: [Function], getLastLogin: [Function] }
              │
              └──> [Closure Scope]
                    - name: "ahmed"
                    - role: "guest"
                    - lastLogin: Date object
```

Privacy in Action:

```
console.log(user1.name); // undefined - private!
console.log(user1.lastLogin); // undefined - private!

// The only way to access is through methods
user1.login();
var loginTime = user1.getLastLogin();
console.log(loginTime); // Works!
```

6.3 Advantages of Factory Pattern

1. No Need for `new` Keyword:

```
// Factory - no 'new' needed
var user = createUser("John", "admin");

// vs Constructor - 'new' required
var user = new User("John", "admin");
// If you forget 'new', things break!
```

2. Flexibility in Return Value:

```
function createProduct(type) {
    if (type === "book") {
        return {
            type: "book",
            read: function() { console.log("Reading..."); }
        };
    } else if (type === "video") {
        return {
            type: "video",
            play: function() { console.log("Playing..."); }
        };
    }
}

var book = createProduct("book");
var video = createProduct("video");

book.read(); // "Reading..."
video.play(); // "Playing..."
```

3. Easy Object Composition:

```
function createLogger(name) {
    return {
        log: function(message) {
            console.log("[" + name + "] " + message);
        }
    };
}

function createUser(name) {
    var logger = createLogger(name);

    return {
        name: name,
        login: function() {
            logger.log("Logged in");
        },
        logout: function() {
            logger.log("Logged out");
        }
    };
}

var user = createUser("Alice");
user.login(); // "[Alice] Logged in"
user.logout(); // "[Alice] Logged out"
```

4. Private Variables and Methods:

```
function createBankAccount(initialBalance) {
    var balance = initialBalance; // Private

    function isValidAmount(amount) { // Private method
        return amount > 0 && typeof amount === 'number';
    }

    return {
        deposit: function(amount) {
            if (isValidAmount(amount)) {
                balance += amount;
                return "Deposited: " + amount;
            }
            return "Invalid amount";
        },
        getBalance: function() {
            return balance;
```

```
        }
    };
}

var account = createBankAccount(100);
console.log(account.deposit(50)); // "Deposited: 50"
console.log(account.getBalance()); // 150
console.log(account.balance); // undefined - private!
console.log(account.isValidAmount); // undefined - private method!
```

6.4 Disadvantages of Factory Pattern

1. Memory Inefficiency:

```
function createProduct(name, price) {
    return {
        name: name,
        price: price,
        discount: function(rate) { // New function for EACH object!
            return this.price * rate;
        }
    };
}

var p1 = createProduct("Book", 100);
var p2 = createProduct("Pen", 50);

console.log(p1.discount === p2.discount); // false
// Each object has its own copy of the discount function
```

Problem: If you create 1000 products, you have 1000 copies of the `discount` function in memory.

2. No Prototype Chain:

```
var p1 = createProduct("Book", 100);
console.log(p1 instanceof createProduct); // false
// Can't use instanceof for type checking
```

3. Methods Not Shared: Each object gets its own copy of every method:

```
function createUser(name) {
    return {
        name: name,
```

```
        greet: function() { // New function instance each time
            console.log("Hello, " + this.name);
        }
    };
}

var users = [];
for (var i = 0; i < 1000; i++) {
    users.push(createUser("User" + i));
}
// 1000 separate greet functions in memory!
```

6.5 When to Use Factory Pattern

Good Use Cases:

1. When you need private data/methods
2. When creating a small number of objects
3. When you want to return different types based on parameters
4. When you don't need instanceof checks
5. For creating configuration objects
6. Module pattern implementations

Bad Use Cases:

1. Creating thousands of objects (memory inefficient)
2. When you need prototype inheritance
3. When instanceof checks are important
4. Performance-critical applications with many objects

6.6 Hybrid Approach - Factory + Prototype

Combining Benefits:

```
function createProduct(name, price) {
    var obj = Object.create(productMethods);
    obj.name = name;
    obj.price = price;
    return obj;
}

var productMethods = {
    discount: function(rate) {
        return this.price * rate;
```

```
    },
    display: function() {
        console.log(this.name + ": $" + this.price);
    }
};

var p1 = createProduct("Book", 100);
var p2 = createProduct("Pen", 50);

console.log(p1.discount === p2.discount); // true - shared method!
```

Benefits:

- Still no `new` keyword
- Methods are shared (memory efficient)
- Prototype chain is established

6.7 Factory Pattern vs Constructor Pattern

```
// Factory Pattern
function createUser(name) {
    return {
        name: name,
        greet: function() {
            console.log("Hi, " + this.name);
        }
    };
}
var user1 = createUser("Alice");

// Constructor Pattern
function User(name) {
    this.name = name;
}
User.prototype.greet = function() {
    console.log("Hi, " + this.name);
};
var user2 = new User("Bob");

// Comparison
console.log(user1.greet === user2.greet); // false (factory)
var user3 = new User("Charlie");
console.log(user2.greet === user3.greet); // true (constructor with prototype)
```

Summary:

- **Factory**: Easy, flexible, good for encapsulation, but memory-heavy
- **Constructor**: More efficient memory use, prototype chain, but requires `new`

---

7. Function Constructor Pattern

Concept: Constructor functions are a way to create multiple objects with the same structure using a blueprint (template). They use the `new` keyword and the prototype chain for memory efficiency.

7.1 Basic Constructor Function

Implementation:

```
function Product(name, price, category) {
    this.name = name;
    this.price = price;
    this.category = category;
}

Product.prototype.discount = function() {
    console.log("discount function");
};

var p1 = new Product("Book", 200, "school");
var p2 = new Product("phone", 3000, "electronics");

console.log(p1);
// Product { name: "Book", price: 200, category: "school" }

console.log(p2);
// Product { name: "phone", price: 3000, category: "electronics" }
```

7.2 What Happens with `new` - Deep Dive

When you write `var p1 = new Product("Book", 200, "school");`, JavaScript performs these 4 steps:

Step 1: Create Empty Object

```
var this = {}; // Internally created by JavaScript
```

Step 2: Link to Prototype

```
this.__proto__ = Product.prototype;
// Sets up inheritance chain
```

Step 3: Execute Constructor

```
// The constructor runs with 'this' bound to the new object
this.name = "Book";
this.price = 200;
this.category = "school";
```

Step 4: Return the Object

```
return this; // Implicit return if no explicit return
```

Complete Visualization:

```
Step 1: new Product("Book", 200, "school")
        ↓
Step 2: this = {}
        ↓
Step 3: this.__proto__ = Product.prototype
        ↓
Step 4: Execute constructor:
        this.name = "Book"
        this.price = 200
        this.category = "school"
        ↓
Step 5: return this
        ↓
        p1 = { name: "Book", price: 200, category: "school" }
```

7.3 The Prototype Chain

Understanding Prototypes:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.discount = function(rate) {
    return this.price * rate;
```

```
};

Product.prototype.display = function() {
    console.log(this.name + ": $" + this.price);
};

var p1 = new Product("Book", 200);
var p2 = new Product("Pen", 50);
```

Memory Diagram:

```
p1 ──> { name: "Book", price: 200 }
       ↓ __proto__
       Product.prototype ──> {
           discount: [Function],
           display: [Function]
       }

p2 ──> { name: "Pen", price: 50 }
       ↓ __proto__
       (same Product.prototype as above)
```

Key Insights:

1. Each object (`p1` , `p2`) has its own properties (`name` , `price`)
2. But they SHARE the same methods via `Product.prototype`
3. Methods are not copied - they're accessed via the prototype chain

Proof of Sharing:

```
console.log(p1.discount === p2.discount); // true
// Both reference the exact same function in memory!

console.log(p1.name === p2.name); // false
// Each has its own 'name' property
```

7.4 Property Lookup Mechanism

How JavaScript Finds Properties:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}
```

```
Product.prototype.discount = function(rate) {
    return this.price * rate;
};

var p1 = new Product("Book", 200);
console.log(p1.discount(0.1)); // 20
```

Lookup Process for `p1.discount(0.1)`:

1. **Look in the object itself**:
   ```
   // Does p1 have a 'discount' property?
   // Check: { name: "Book", price: 200, discount: ??? }
   // Not found on p1 directly
   ```
2. **Look in the prototype**:
   ```
   // Follow __proto__ to Product.prototype
   // Check: Product.prototype.discount
   // Found! Use this function
   ```
3. **Execute with correct `this`**:
   ```
   // Call discount function with 'this' = p1
   // this.price = p1.price = 200
   // Return 200 * 0.1 = 20
   ```

Full Lookup Chain:

```
p1.discount(0.1)
  ↓
  1. Look in p1 { name: "Book", price: 200 }
     Not found ↓
  2. Look in p1.__proto__ (Product.prototype)
     Found! ✓ { discount: [Function], display: [Function] }
  3. Execute with this = p1
     return this.price * rate = 200 * 0.1 = 20
```

7.5 Adding Methods - Instance vs Prototype

Instance Methods (BAD - Memory Wasteful):

```
function Product(name, price) {
    this.name = name;
    this.price = price;
```

```
    // DON'T DO THIS!
    this.discount = function(rate) {
        return this.price * rate;
    };
}

var p1 = new Product("Book", 200);
var p2 = new Product("Pen", 50);

console.log(p1.discount === p2.discount); // false
// Each object has its own copy - wasteful!
```

Memory Impact:

```
p1 ──> { name: "Book", price: 200, discount: [Function A] }
p2 ──> { name: "Pen", price: 50, discount: [Function B] }

Function A and Function B are identical but occupy separate memory!
```

Prototype Methods (GOOD - Memory Efficient):

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}

// Add method to prototype (outside constructor)
Product.prototype.discount = function(rate) {
    return this.price * rate;
};

var p1 = new Product("Book", 200);
var p2 = new Product("Pen", 50);

console.log(p1.discount === p2.discount); // true
// Shared method - efficient!
```

Memory Impact:

```
p1 ──> { name: "Book", price: 200 }
        ↓ __proto__
p2 ──> { name: "Pen", price: 50 }
        ↓ __proto__
```

```
        Product.prototype ──> { discount: [Function] }

Only ONE discount function in memory, shared by all instances!
```

7.6 The `constructor` Property

Understanding `constructor`:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}

var p1 = new Product("Book", 200);

console.log(p1.constructor); // [Function: Product]
console.log(p1.constructor === Product); // true
```

How It Works:

- Every function's prototype has a `constructor` property
- It points back to the function itself
- Objects created with `new` inherit this via the prototype chain

Diagram:

```
Product (function)
  ↓ prototype
Product.prototype
  ↓ constructor
Product (back to the function)

p1
  ↓ __proto__
Product.prototype
  ↓ constructor
Product
```

Using `constructor` for Cloning:

```
var p1 = new Product("Book", 200);
var p2 = new p1.constructor("Pen", 50);
```

```
console.log(p2); // Product { name: "Pen", price: 50 }
```

7.7 Checking Instance Type

Using `instanceof`:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}

var p1 = new Product("Book", 200);

console.log(p1 instanceof Product); // true
console.log(p1 instanceof Object); // true (all objects inherit from Object)
console.log(p1 instanceof Array); // false
```

How `instanceof` Works:

```
// p1 instanceof Product checks if:
// Product.prototype exists in p1's prototype chain

p1.__proto__ === Product.prototype? // true
```

7.8 Forgetting `new` - The Problem

What Happens Without `new`:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}

var p1 = Product("Book", 200); // Forgot 'new'!

console.log(p1); // undefined
console.log(window.name); // "Book" - Oops!
console.log(window.price); // 200 - Polluted global scope!
```

Why This Happens:

- Without `new`, `this` refers to `window` (default binding)

- `this.name = "Book"` becomes `window.name = "Book"`
- The function doesn't return anything, so `p1` is `undefined`

Solution 1 - Naming Convention:

```
// Always capitalize constructor functions
function Product() {} // Constructor
function createProduct() {} // Factory
```

Solution 2 - Defensive Code:

```
function Product(name, price) {
    if (!(this instanceof Product)) {
        return new Product(name, price);
    }

    this.name = name;
    this.price = price;
}

var p1 = Product("Book", 200); // Works even without 'new'
```

Solution 3 - ES6 Classes (modern approach):

```
class Product {
    constructor(name, price) {
        this.name = name;
        this.price = price;
    }

    discount(rate) {
        return this.price * rate;
    }
}

var p1 = new Product("Book", 200); // Must use 'new'
var p2 = Product("Pen", 50); // TypeError - forces you to use 'new'
```

7.9 Inheritance with Constructors

Basic Inheritance:

```
function Product(name, price) {
    this.name = name;
    this.price = price;
}

Product.prototype.display = function() {
    console.log(this.name + ": $" + this.price);
};

function Book(name, price, author) {
    Product.call(this, name, price); // Call parent constructor
    this.author = author;
}

// Set up inheritance
Book.prototype = Object.create(Product.prototype);
Book.prototype.constructor = Book;

Book.prototype.showAuthor = function() {
    console.log("By " + this.author);
};

var book = new Book("JS Guide", 50, "John Doe");
book.display(); // "JS Guide: $50" (inherited from Product)
book.showAuthor(); // "By John Doe" (Book's own method)
```

Inheritance Chain:

```
book
  ↓ __proto__
Book.prototype
  ↓ __proto__
Product.prototype
  ↓ __proto__
Object.prototype
  ↓ __proto__
null
```

7.10 Constructor Pattern vs Factory Pattern

Constructor Pattern:

```
function Product(name, price) {
    this.name = name;
```

```
    this.price = price;
}

Product.prototype.discount = function(rate) {
    return this.price * rate;
};

var p1 = new Product("Book", 200);
```

Pros:

- Memory efficient (shared methods via prototype)
- Works with `instanceof`
- Standard OOP pattern
- Prototype inheritance

Cons:

- Requires `new` keyword (easy to forget)
- No private variables (without closures)
- `this` binding can be confusing

Factory Pattern:

```
function createProduct(name, price) {
    return {
        name: name,
        price: price,
        discount: function(rate) {
            return this.price * rate;
        }
    };
}

var p1 = createProduct("Book", 200);
```

Pros:

- No `new` keyword needed
- Easy to create private variables
- Flexible return values
- Simpler for beginners

Cons:

- Memory inefficient (methods not shared)
- No `instanceof` support
- No prototype inheritance
- Each object is standalone

When to Use Each:

- **Constructor**: Large-scale applications, need inheritance, many instances
- **Factory**: Small number of objects, need privacy, simple structures

---

8. `this` in Modules and Strict Mode

8.1 ES6 Modules and `this`

Concept: In ES6 modules, the top-level `this` is `undefined`, not the global object.

Regular Script File:

```
// script.js (not a module)
console.log(this); // window (in browsers)

function test() {
    console.log(this); // window (default binding)
}
test();
```

ES6 Module:

```
// module.js (ES6 module)
console.log(this); // undefined

function test() {
    console.log(this); // undefined (strict mode by default)
}
test();
```

Why This Happens:

- ES6 modules automatically run in strict mode

- In strict mode, top-level `this` is `undefined`
- This prevents accidental global pollution

Deep Dive:

```
// Regular script
var x = 10;
console.log(this.x); // 10 (this = window, window.x = 10)

// ES6 module
var x = 10;
console.log(this.x); // TypeError: Cannot read property 'x' of undefined
console.log(x); // 10 (still accessible, but not via this)
```

Accessing Global Object in Modules:

```
// If you need the global object in a module
console.log(globalThis); // Standardized way (ES2020)
console.log(window); // Browser-specific
console.log(global); // Node.js-specific
```

8.2 Strict Mode and `this`

Concept: Strict mode changes how `this` behaves in several ways, primarily affecting default binding.

Enabling Strict Mode:

```
'use strict';

// Or at function level
function myFunc() {
    'use strict';
    // strict mode only in this function
}
```

Default Binding Changes:

```
// Non-strict mode
function test() {
    console.log(this); // window
}
test();
```

```
// Strict mode
'use strict';
function test() {
    console.log(this); // undefined
}
test();
```

Why This is Important:

```
// Non-strict mode - dangerous!
function addProperty() {
    this.name = "John"; // Accidentally adds to window
}
addProperty();
console.log(window.name); // "John" - Oops!

// Strict mode - safer
'use strict';
function addProperty() {
    this.name = "John"; // TypeError: Cannot set property of undefined
}
addProperty(); // Error prevents accidental global pollution
```

8.3 Other Strict Mode Effects

1. Prevents Accidental Globals:

```
'use strict';

function test() {
    x = 10; // ReferenceError: x is not defined
    // Without strict mode, this would create window.x
}
```

2. Prevents Deleting Variables:

```
'use strict';

var x = 10;
delete x; // SyntaxError
```

3. Duplicate Parameter Names:

```
'use strict';

function test(a, a, b) { // SyntaxError
    console.log(a, b);
}
```

4. Octal Literals:

```
'use strict';

var num = 010; // SyntaxError
// Octal notation not allowed
```

5. Assignment to Read-Only Properties:

```
'use strict';

var obj = {};
Object.defineProperty(obj, 'x', {
    value: 1,
    writable: false
});

obj.x = 2; // TypeError
// Without strict mode, this fails silently
```

8.4 Strict Mode and Object Methods

Implicit Binding Still Works:

```
'use strict';

var obj = {
    name: "Test",
    getName: function() {
        console.log(this.name); // Works fine
    }
};

obj.getName(); // "Test" - implicit binding not affected
```

Lost Binding Becomes `undefined`:

```
'use strict';

var obj = {
    name: "Test",
    getName: function() {
        console.log(this); // undefined (in strict mode)
    }
};

var func = obj.getName;
func(); // undefined - not window!
```

8.5 Strict Mode in Different Contexts

Global Strict Mode:

```
'use strict';

console.log(this); // window (in browsers)
// Top-level 'this' is still window in non-module scripts

function test() {
    console.log(this); // undefined
}
test();
```

Function-Level Strict Mode:

```
function nonStrict() {
    console.log(this); // window
}

function strict() {
    'use strict';
    console.log(this); // undefined
}

nonStrict(); // window
strict(); // undefined
```

Mixing Strict and Non-Strict:

```
function outer() {
    console.log(this); // window

    function inner() {
        'use strict';
        console.log(this); // undefined
    }

    inner();
}

outer();
```

8.6 Module Pattern with Strict Mode

Combining Modules and Strict Mode:

```
// module.js
'use strict'; // Optional - modules are strict by default

export function createCounter() {
    let count = 0; // Private variable

    return {
        increment() {
            count++;
            console.log(this); // Depends on how it's called
        },
        getCount() {
            return count;
        }
    };
}

// main.js
import { createCounter } from './module.js';

const counter = createCounter();
counter.increment(); // 'this' is the counter object
```

8.7 Practical Implications

1. Constructor Functions:

```
'use strict';

function Product(name) {
    this.name = name;
}

var p1 = new Product("Book"); // Works - 'new' creates object
var p2 = Product("Pen"); // TypeError - 'this' is undefined
```

2. Event Handlers:

```
'use strict';

button.addEventListener('click', function() {
    console.log(this); // Still the button element
    // Event handlers explicitly set 'this'
});
```

3. Arrow Functions:

```
'use strict';

const obj = {
    name: "Test",
    getName: () => {
        console.log(this); // undefined (lexical 'this' from module)
    }
};

obj.getName(); // undefined
```

8.8 Browser vs Node.js

Browser:

```
// Non-module script
console.log(this === window); // true

// ES6 module
console.log(this); // undefined
console.log(window); // Window object (still accessible)
```

Node.js:

```
// Regular file
console.log(this === global); // false
console.log(this === module.exports); // true

// ES6 module
console.log(this); // undefined
console.log(global); // Global object (still accessible)
```

8.9 Best Practices

1. Always Use Strict Mode:

```
'use strict';
// Catches errors early, prevents bad patterns
```

2. Use ES6 Modules:

```
// Automatic strict mode, better encapsulation
export function myFunction() {
    // ...
}
```

3. Be Explicit About Context:

```
'use strict';

function myFunction() {
    console.log(this); // Makes undefined clear
}

myFunction.call(obj); // Explicit context
```

4. Use Arrow Functions for Callbacks:

```
'use strict';

const obj = {
    name: "Test",
    delayedGreet() {
        setTimeout(() => {
            console.log(this.name); // Lexical 'this' works in strict mode
        }, 1000);
```

```
    }
};
```

8.10 Summary Table

Context	Non-Strict	Strict Mode	ES6 Module
Top-level `this`	window	window	undefined
Function `this` (default)	window	undefined	undefined
Method `this`	object	object	object
Constructor without `new`	window (danger!)	undefined (error)	undefined (error)
Arrow function `this`	Lexical	Lexical	Lexical

Key Takeaways

1. `this` **Binding**: Determined by call-site, not definition location (except arrow functions)
2. **Closures**: Functions remember their outer scope even after it's gone
3. **Factory Pattern**: Simple object creation, good for privacy, memory-intensive
4. **Constructor Pattern**: Memory-efficient via prototypes, standard OOP approach
5. **Strict Mode**: Prevents errors, changes `this` behavior, safer code
6. **Modules**: Automatic strict mode, `this` is `undefined` at top level

These concepts form the foundation of advanced JavaScript programming and are essential for understanding modern frameworks and libraries.

Abdullah Ali

Contact : +201012613453