JavaScript Fundamentals - Part 2

> 🎓 **CS Perspective:** This section dives into **functions as first-class citizens**—a core concept from functional programming where functions can be assigned to variables, passed as arguments, and returned from other functions. You'll explore different function declaration syntaxes and their implications for **hoisting** (compile-time vs. runtime behavior) and **lexical scoping**. The comparison of function declarations, expressions, and arrow functions illustrates how JavaScript handles the `this` binding and closure creation differently based on syntax.

📌 Strict Mode

Activates stricter parsing and error handling:

```
'use strict'; // Must be first statement in script
```

Benefits

- Catches common coding mistakes
- Prevents accidental global variables
- Reserves keywords for future JS versions (`interface`, `private`)

```
'use strict';
let hasLicense = false;
hasLicence = true; // ❌ ReferenceError (typo caught!)
```

📌 Functions

Reusable blocks of code that perform a specific task.

```
// Define function
function logger() {
  console.log('My name is Jonas');
}

// Call / invoke / run function
logger();
logger();
```

Functions with Parameters and Return

```
function fruitProcessor(apples, oranges) {
  const juice = `Juice with ${apples} apples and ${oranges} oranges.`;
  return juice;  // Output value
}

const result = fruitProcessor(5, 3); // Arguments
console.log(result);
```

Term	Description
Parameters	Placeholders in function definition
Arguments	Actual values passed when calling
Return	Output value from function

📌 Function Types

1. Function Declaration

```
function calcAge(birthYear) {
  return 2037 - birthYear;
}
```

✅ Can be called **before** declaration (hoisting)

2. Function Expression

```
const calcAge = function(birthYear) {
  return 2037 - birthYear;
};
```

❌ Cannot be called before declaration

3. Arrow Function (ES6)

```
// One-liner (implicit return)
const calcAge = birthYear => 2037 - birthYear;

// Multiple parameters
const calcAge = (birthYear, firstName) => {
  const age = 2037 - birthYear;
  return `${firstName} is ${age} years old`;
};
```

Comparison Table

Feature	Declaration	Expression	Arrow
Hoisted	✅ Yes	❌ No	❌ No
`this` keyword	✅ Own	✅ Own	❌ Inherits
`arguments`	✅ Yes	✅ Yes	❌ No
Syntax	`function name()`	`const name = function()`	`const name = () =>`

📌 Functions Calling Other Functions

```
function cutPieces(fruit) {
  return fruit * 4;
}

function fruitProcessor(apples, oranges) {
  const applePieces = cutPieces(apples);
  const orangePieces = cutPieces(oranges);
  return `Juice with ${applePieces} apple pieces and ${orangePieces}
orange pieces`;
}

console.log(fruitProcessor(2, 3)); // 8 apple pieces, 12 orange pieces
```

📌 Arrays

Ordered collection of values.

Creating Arrays

```
// Literal syntax (preferred)
const friends = ['Michael', 'Steven', 'Peter'];

// Constructor
const years = new Array(1991, 1984, 2008);
```

Accessing Elements

```
console.log(friends[0]);    // 'Michael' (first)
console.log(friends[2]);    // 'Peter' (third)
console.log(friends.length); // 3
```

```
// Last element
console.log(friends[friends.length - 1]); // 'Peter'
```

Mutating Arrays

```
friends[2] = 'Jay'; // Replace element
// ✅ Works even with const (array reference unchanged)

friends = ['Bob']; // ❌ Error (can't reassign const)
```

Arrays Can Hold Mixed Types

```
const jonas = ['Jonas', 'Schmedtmann', 46, 'teacher', friends];
```

📌 Array Methods

Adding Elements

Method	Description	Returns
`push(el)`	Add to **end**	New length
`unshift(el)`	Add to **beginning**	New length

```
const friends = ['Michael', 'Steven'];
friends.push('Jay');      // ['Michael', 'Steven', 'Jay']
friends.unshift('John');  // ['John', 'Michael', 'Steven', 'Jay']
```

Removing Elements

Method	Description	Returns
`pop()`	Remove from **end**	Removed element
`shift()`	Remove from **beginning**	Removed element

```
friends.pop();    // Returns 'Jay'
friends.shift();  // Returns 'John'
```

Searching

Method	Description	Returns
`indexOf(el)`	Find index	Index or `-1`
`includes(el)`	Check existence	`true` / `false`

```
friends.indexOf('Steven');  // 1
friends.indexOf('Bob');     // -1 (not found)

friends.includes('Steven'); // true
friends.includes('Bob');    // false
```

⚠ `includes()` uses **strict equality** (`===`)

📌 Objects

Unordered collection of key-value pairs.

```
const jonas = {
  firstName: 'Jonas',
  lastName: 'Schmedtmann',
  age: 46,
  job: 'teacher',
  friends: ['Michael', 'Peter', 'Steven']
};
```

Dot vs Bracket Notation

```
// Dot notation
console.log(jonas.lastName);     // 'Schmedtmann'

// Bracket notation
console.log(jonas['lastName']);  // 'Schmedtmann'

// Bracket with expression
const key = 'Name';
console.log(jonas['first' + key]); // 'Jonas'
```

When to Use Each

Dot Notation	Bracket Notation
Simple property names	Computed property names
Clean syntax	Dynamic access

Dot Notation	Bracket Notation
`jonas.age`	`jonas[variable]`

Adding Properties

```
jonas.location = 'Portugal';
jonas['twitter'] = '@jonas';
```

📌 Object Methods

Functions inside objects.

```
const jonas = {
  firstName: 'Jonas',
  birthYear: 1991,

  // Method
  calcAge: function() {
    this.age = 2037 - this.birthYear;
    return this.age;
  }
};

jonas.calcAge();     // Calculates and stores age
console.log(jonas.age); // 46
```

The `this` Keyword

- Inside a method, `this` refers to the **object calling the method**
- Allows accessing other properties of the same object

```
getSummary: function() {
  return `${this.firstName} is ${this.calcAge()} years old`;
}
```

📌 The for Loop

Repeat code a specific number of times.

```
// for (counter; condition; increment)
for (let rep = 1; rep <= 10; rep++) {
```

```
  console.log(`Repetition ${rep}`);
}
```

Looping Through Arrays

```
const arr = ['Jonas', 46, 'teacher'];

for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

Building New Arrays

```
const years = [1991, 2007, 1969];
const ages = [];

for (let i = 0; i < years.length; i++) {
  ages.push(2037 - years[i]);
}
// ages = [46, 30, 68]
```

📌 continue and break

continue

Skip current iteration, continue to next:

```
for (let i = 0; i < arr.length; i++) {
  if (typeof arr[i] !== 'string') continue; // Skip non-strings
  console.log(arr[i]);
}
```

break

Exit loop entirely:

```
for (let i = 0; i < arr.length; i++) {
  if (typeof arr[i] === 'number') break; // Stop at first number
  console.log(arr[i]);
}
```

📌 Looping Backwards

```
const arr = ['a', 'b', 'c', 'd'];

for (let i = arr.length - 1; i >= 0; i--) {
  console.log(arr[i]); // d, c, b, a
}
```

📌 Nested Loops

```
for (let exercise = 1; exercise <= 3; exercise++) {
  console.log(`--- Exercise ${exercise} ---`);

  for (let rep = 1; rep <= 5; rep++) {
    console.log(`Rep ${rep}`);
  }
}
```

📌 The while Loop

Use when you **don't know** how many iterations needed.

```
let rep = 1;
while (rep <= 10) {
  console.log(`Rep ${rep}`);
  rep++;
}
```

Example: Random Dice

```
let dice = Math.trunc(Math.random() * 6) + 1;

while (dice !== 6) {
  console.log(`Rolled: ${dice}`);
  dice = Math.trunc(Math.random() * 6) + 1;
}
```

📌 for vs while

for	while

for	while
Known iteration count	Unknown iteration count
Counter built-in	Manual counter
`for (let i = 0; i < 10; i++)`	`while (condition)`
Looping arrays	Condition-based loops

🧪 Quick Reference Cheatsheet

```
// Strict mode
'use strict';

// Function types
function fn() {}                    // Declaration
const fn = function() {};           // Expression
const fn = () => {};                // Arrow

// Arrays
const arr = [1, 2, 3];
arr.push(4);          // Add end
arr.unshift(0);       // Add start
arr.pop();            // Remove end
arr.shift();          // Remove start
arr.indexOf(2);       // Find index
arr.includes(2);      // Check exists

// Objects
const obj = { key: 'value' };
obj.key               // Dot notation
obj['key']            // Bracket notation
this                  // Reference to current object

// Loops
for (let i = 0; i < arr.length; i++) {}
while (condition) {}
continue;             // Skip iteration
break;                // Exit loop
```

✅ Exam Tips

1. **Arrow functions** don't have their own `this`
2. `push` returns **new length**, not the array
3. `includes()` uses **strict equality**
4. Arrays declared with `const` can still be **mutated**
5. Use **bracket notation** for dynamic property access
6. `this` inside a method = the **object calling** the method

7. `continue` skips, `break` exits
8. `while` loop needs manual counter increment
9. Always use `'use strict';` at the top of your scripts
10. Function declarations are **hoisted**, expressions are **not**