

PEDAGOGICAL APPROACHES TO TEACHING JAVASCRIPT FUNCTIONS: A FOCUS ON ARROW, NAMED, ANONYMOUS, AND CALLBACK FUNCTIONS

ABORDĂRI PEDAGOGICE ÎN PREDAREA FUNCȚIILOR JAVASCRIPT: UN FOCUS PE FUNCȚIILE SĂGEATĂ, NUMITE, ANONIME ȘI DE CALLBACK

CZU: 004.43:378.147

<https://doi.org/10.59295/spd2025e.118>

Vadim SIRKELI

ORCID: 0000-0002-2479-4218

E-mail: vadim.sirkeli@gmail.com

Moldova State University, Chisinau, Republic of Moldova

Iulia BORIS

ORCID: 0000-0003-4523-2182

E-mail: borisiulia@yahoo.com

Moldova State University, Chisinau, Republic of Moldova

Denis NIKA

ORCID: 0000-0002-3082-3118

E-mail: dlnika@yahoo.com

Moldova State University, Chisinau, Republic of Moldova

Sumar. Funcțiile reprezintă fundamentul codului modular și reutilizabil în JavaScript, însă diversitatea sintaxelor și paradigmelor acestora — inclusiv funcțiile numite, anonime, săgeată și de callback — prezintă adesea provocări pedagogice semnificative pentru cadrele didactice și obstacole de învățare pentru studenți. Această lucrare explorează strategii pedagogice eficiente concepute pentru a demistifica aceste tipuri variate de funcții. Bazându-ne pe principii din teoria sarcinii cognitive, constructivism și sprijin pedagogic (scaffolding), propunem o abordare structurată care pune accent pe analogii, exemple practice și aplicare directă. Prin analizarea caracteristicilor unice și a cazurilor de utilizare ale fiecărui tip de funcție, ne propunem să oferim cadrelor didactice metode aplicabile pentru a îmbunătăți înțelegerea studenților; a reduce concepțiile greșite frecvente și a încuraja o înțelegere mai profundă a capacităților funcționale ale JavaScript.

Cuvinte-cheie: evaluare, funcții de callback, educație în informatică, funcții, JavaScript.

1. INTRODUCTION

In the realm of programming, functions stand as indispensable constructs, serving as the fundamental building blocks that enable the creation of modular, reusable, and abstract code. They encapsulate specific logic, allowing developers to break down complex

problems into manageable units, thereby enhancing code readability, maintainability, and scalability. In JavaScript, this significance is amplified by the language's dynamic, asynchronous, and event-driven nature. Functions in JavaScript are first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions, underpinning powerful patterns like callbacks, closures, and higher-order functions. This inherent flexibility makes a thorough understanding of functions crucial for any aspiring JavaScript developer.

Despite their foundational importance, teaching JavaScript functions presents a unique set of challenges. The language offers multiple syntaxes for defining functions, from traditional function declarations and expressions to the more modern arrow functions (introduced in ES6). Furthermore, the distinct roles of named functions, anonymous functions, and callback functions can often blur in a student's mind, leading to confusion regarding their appropriate use cases, scope, and particularly, the often-misunderstood this context. Common student difficulties include discerning when to use an anonymous function versus a named one, grappling with the lexical this binding of arrow functions, and comprehending the asynchronous flow inherent in callback patterns. These complexities necessitate targeted and deliberate pedagogical strategies to ensure effective knowledge transfer and skill development.

This paper aims to provide a comprehensive exploration of effective pedagogical approaches for teaching the various types of JavaScript functions: named, anonymous, arrow, and callback functions. Our primary objective is to equip educators with practical strategies and conceptual frameworks to address common student difficulties and foster a robust understanding of these core JavaScript concepts. By dissecting each function type, highlighting its unique characteristics, and proposing specific teaching methodologies grounded in educational theories, we seek to enhance the learning experience and improve student proficiency in JavaScript programming.

2. METHODS

Effective programming education is not merely about presenting syntax; it requires a deep understanding of how students learn and process complex information. This article's pedagogical recommendations are rooted in established educational theories that offer insights into optimizing the learning process for programming concepts.

2.1. Cognitive Load Theory in Programming Education

Cognitive Load Theory (CLT), proposed by Sweller [1], posits that working memory has a limited capacity. When instructional design overloads this capacity, learning can be hindered. In programming, introducing multiple function types with subtle syntactical and behavioral differences can significantly increase extraneous cognitive load — the mental effort imposed by the way information is presented, rather than by the information itself. To mitigate this, pedagogical strategies should aim to reduce unnecessary

complexity, present information clearly, and provide sufficient practice for students to develop automated schemas. For instance, gradually introducing function types, starting with the simplest before moving to more complex ones, can prevent cognitive overload. Visual aids, consistent terminology, and breaking down complex examples into smaller, digestible parts are also critical for managing cognitive load.

2.2. Constructivism and Active Learning

Constructivism emphasizes that learners actively construct their own understanding and knowledge of the world through experiencing things and reflecting on those experiences. In programming education, this translates to the paramount importance of active learning. Students don't truly grasp function concepts by passively listening to lectures; they learn by writing code, debugging errors, experimenting with different syntaxes, and solving problems. This approach necessitates hands-on exercises, coding challenges, and project-based learning where students apply various function types in meaningful contexts. Collaborative learning, such as pair programming, further enhances this by allowing students to discuss, explain, and justify their coding decisions, solidifying their understanding.

2.3. Scaffolding in Programming Instruction

Scaffolding, a concept derived from Vygotsky's Zone of Proximal Development [2], involves providing temporary support to learners as they acquire new skills, gradually withdrawing that support as they become more proficient. In teaching JavaScript functions, scaffolding is crucial. An educator might start by providing highly structured code templates for named functions, then progressively introduce more complex scenarios where students need to independently choose and implement anonymous or arrow functions. This phased approach builds confidence and competence, allowing students to tackle increasingly challenging problems with diminishing external assistance. For instance, initial exercises might involve simple function definitions, followed by problems requiring functions as arguments, and finally, complex asynchronous scenarios.

3. PEDAGOGICAL STRATEGIES FOR TEACHING JAVASCRIPT FUNCTION TYPES

A structured and progressive approach is essential when introducing the diverse landscape of JavaScript functions [3]. This section outlines specific pedagogical strategies for named, anonymous, arrow, and callback functions.

3.1. Named Functions: The Foundation

Named functions are defined using the "function" keyword, followed by a name, a list of parameters, and the function body. They are hoisted, meaning they can be called

before their declaration in the code as shown in Example 1.

Example 1:

```
function greet(name) {  
    return `Welcome, ${name}!`;  
}  
greet("USM"); // Returns "Welcome, USM!"
```

Pedagogical approach: Introduce named functions as the fundamental building blocks. Use the analogy of a “named recipe” or a “defined procedure” that can be repeatedly followed. Emphasize their clarity, reusability, and ease of debugging due to their explicit names which appear in stack traces. Begin with simple, tangible examples like basic arithmetic operations (**add(a, b)**), string manipulations (**reverseString(str)**), or simple DOM interactions (**showElement(id)**). Encourage students to “write a function that...” solves a specific, small problem. This is also the ideal point to introduce core concepts like function scope and parameters.

3.2. Anonymous Functions: Flexibility and Immediacy

Anonymous functions are functions without a name. They are often used as expressions, assigned to variables, or passed as arguments to other functions (Example 2).

Example 2:

```
const add = function(a, b) {  
    return a + b;  
};  
add(2, 7); // Returns 9
```

Present anonymous functions as “one-off” instructions or procedures that don’t require a formal name because they are used immediately or passed directly. The immediate context for introducing anonymous functions should be scenarios where a function is used once or passed as an argument. Excellent examples include **setTimeout()** or array methods like **forEach** or **map**:

Example 3:

```
setTimeout(function() {  
    console.log("Delayed message!");  
}, 1000); // Logs after 1 second
```

Example 4:

```
const numbers = [1, 2, 3, 4, 5];  
numbers.forEach(function(num) {  
    console.log(num * 2);  
}); // Logs 2, 4, 6, 8, 10
```

Discuss the trade-offs between named and anonymous functions, focusing on readability versus conciseness. When an anonymous function becomes complex, a named function might be more appropriate for clarity.

3.3. Arrow Functions: Modern Conciseness and Lexical “this”

Introduced in ECMAScript 2015 (ES6), arrow functions provide a more concise syntax for writing function expressions [3]. Their key characteristic is their lexical “this” binding, meaning “this” is bound to the context in which the arrow function is defined, rather than the context in which it is called.

Example 5:

```
const multiply = (a, b) => a * b; // Implicit return for single expression
const
logMessage = () => console.log('Hello, logging'); // No parameters, no
explicit return
const user = {
  name: 'USM',
  greet: function() { // Traditional function: 'this' refers to 'user'
    setTimeout(function() {
      console.log('Welcome, ${this.name}'); // 'this' is global or undefined in
strict mode
    }, 100);
  },
  greetArrow: function() { // Arrow function: 'this' is lexically
bound to 'user'
    setTimeout(() => {
      console.log('Welcome, ${this.name}'); // 'this' refers to 'user'
    }, 100);
  }
};
user.greet(); // Might log "Welcome, undefined" or "Welcome, [global
name]"
user.greetArrow(); // Logs "Welcome, USM"
```

3.4. Callback Functions: Asynchronous and Event-Driven Programming Core

A callback function is a function passed as an argument to another function, to be executed later, after some operation has completed. They are fundamental to JavaScript's asynchronous nature and event handling. Callbacks can be any of the function types discussed (named, anonymous, arrow).

Example 6:

```
function fetchData(url, callback) {
  console.log('Fetching data from: ${url}');
  // Simulate an asynchronous operation (e.g., network request)
  setTimeout(() => {
    const data = `Processed data for ${url}`;
```

```
callback(data); // Execute the callback with the data
    }, 1500);
}
// Using an anonymous function as a callback
fetchData('api.com/users', function(result) {
    console.log('Received users data:', result);
});
// Using an arrow function as a callback
fetchData('api.com/posts', (result) => {
    console.log('Received posts data:', result);
});
```

Pedagogical approach: use the analogy of “leaving a message for someone to act on later” or “hiring someone to do a task and report back when finished”. Emphasize that the calling function dictates when the callback is executed, not necessarily immediately. Explain their necessity in handling asynchronous operations (e.g., network requests, timers) and event-driven programming (e.g., user clicks, key presses). Visually demonstrate how control is passed to an asynchronous operation, and then, at a later point, the callback “takes control” to process the result. Live coding is crucial here. Start with **setTimeout**, then move to **addEventListener** for DOM events. Build simple interactive elements where user actions trigger callback functions. introduce the concept of “callback hell” (nested callbacks) as a natural consequence of deeply asynchronous operations, providing a motivational context for more advanced topics like **Promises** and **Async/Await**, without diving into them in this article.

4. CONCLUSION

Mastering JavaScript functions is a critical milestone for any aspiring developer. The pedagogical strategies outlined in this article underscore the importance of a structured, scaffolded approach that systematically introduces named, anonymous, arrow, and callback functions. By employing accessible analogies, providing clear and progressive examples, and emphasizing hands-on practice, educators can effectively navigate the complexities inherent in these diverse function types. The focus on managing cognitive load, promoting active learning, and leveraging robust scaffolding ensures that students build a strong foundational understanding. These strategies offer a roadmap for educators to demystify JavaScript functions, thereby improving student comprehension and significantly reducing common misconceptions related to syntax, scope, and the critical “this” keyword. By integrating comparative analysis and encouraging best practices, educators can empower students to not only understand how functions work but also to make informed decisions about their application in real-world scenarios. Diversifying teaching methods to include visual aids, interactive tools, and collaborative learning further enhances the learning experience. While this article proposes effective pedagogical approaches, empirical studies are warranted to rigorously evaluate the impact of specific

interventions on student learning outcomes. Future research could investigate the effectiveness of particular visual programming tools in clarifying complex concepts like `this` binding, analyze the long-term retention of function knowledge across different teaching methodologies, or explore the challenges and best practices in teaching advanced functional programming paradigms in JavaScript, building upon the foundational understanding of the function types discussed herein. Such research would provide valuable data to further refine and optimize JavaScript programming education.

REFERENCES:

1. SWELLER, J. Cognitive Load During Problem Solving: Effects on Learning. În: *Cognitive Science*. 1988. vol. 12, no. 2, p. 257-285.
2. VYGOTSKY, L.S. *Mind in Society: The Development of Higher Psychological Processes*. Cambridge, MA: Harvard University Press. 1978. 159 p. ISBN 978-0674576292.
3. ECMAScript 2015 Language Specification. ECMA International. Available: <https://262.ecma-international.org/6.0/> [Accessed: 23.06.2025]

***N.B.** The authors acknowledge the financial support from the Ministry of Education and Research of the Republic of Moldova (research program no. 011208).*