JAVASCRIPT INTERVIEW QUESTIONS & SOLUTIONS

1. Explain var, let, and const differences.

Answer:

Feature	var	let	const
Scope	Function/Global	Block	Block
Hoisting	Yes (undefined)	Yes (TDZ)	Yes (TDZ)
Reassignment	Yes	Yes	No
Redeclaration	Yes	No	No

Example:

javascript

```javascript
// var - function scoped
function example() {
  if (true) {
    var x = 1;
  }
  console.log(x); // 1 (accessible outside block)
}

// let - block scoped
function example2() {
  if (true) {
    let y = 1;
  }
  console.log(y); // ReferenceError: y is not defined
}

// const - block scoped, cannot reassign
const z = 1;
z = 2; // TypeError: Assignment to constant variable

// But objects/arrays can be mutated
const obj = { name: 'John' };
obj.name = 'Jane'; // This works
obj.age = 25; // This works
```

2. What are closures and provide an example?

Answer: A closure is a function that has access to variables in its outer (enclosing) scope even after the outer function has returned.

Example:

javascript
```javascript
function outerFunction(x) {
  // Outer function's variable
  let outerVariable = x;

  function innerFunction(y) {
    // Inner function has access to outerVariable
    console.log(outerVariable + y);
  }

  return innerFunction;
}

const addFive = outerFunction(5);
addFive(3); // Outputs: 8

// Practical example - Counter
function createCounter() {
  let count = 0;

  return {
    increment: () => ++count,
    decrement: () => --count,
    getCount: () => count
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.getCount()); // 2
```

3. Explain 'this' keyword in JavaScript.

Answer: The `this` keyword refers to the object that is executing the current function.

Different Contexts:

javascript
```javascript
// Global context
console.log(this); // Window object (browser) or global (Node.js)

// Object method
```

```javascript
const person = {
  name: 'John',
  greet: function() {
    console.log(this.name); // 'John'
  }
};

// Arrow functions don't have their own 'this'
const person2 = {
  name: 'Jane',
  greet: () => {
    console.log(this.name); // undefined (inherits from global)
  }
};

// Constructor function
function Person(name) {
  this.name = name;
  this.greet = function() {
    console.log(this.name);
  };
}

const john = new Person('John');
john.greet(); // 'John'

// call, apply, bind
const sayHello = function() {
  console.log(`Hello, ${this.name}`);
};

sayHello.call({name: 'Alice'}); // 'Hello, Alice'
sayHello.apply({name: 'Bob'}); // 'Hello, Bob'
const boundFunc = sayHello.bind({name: 'Charlie'});
boundFunc(); // 'Hello, Charlie'
```

4. What are Promises and async/await?

Answer: Promises handle asynchronous operations, providing an alternative to callbacks.

Promise States:

- Pending - Initial state
- Fulfilled - Operation completed successfully
- Rejected - Operation failed

Example:

javascript

```javascript
// Creating a Promise
function fetchUserData(userId) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (userId > 0) {
        resolve({ id: userId, name: 'John Doe' });
      } else {
        reject(new Error('Invalid user ID'));
      }
    }, 1000);
  });
}

// Using Promise with .then()
fetchUserData(1)
  .then(user => {
    console.log('User:', user);
    return fetchUserData(2); // Chain another promise
  })
  .then(user2 => {
    console.log('User 2:', user2);
  })
  .catch(error => {
    console.error('Error:', error.message);
  });

// Using async/await
async function getUsers() {
  try {
    const user1 = await fetchUserData(1);
    const user2 = await fetchUserData(2);
    console.log('Users:', user1, user2);
  } catch (error) {
    console.error('Error:', error.message);
  }
}

// Promise.all for parallel execution
async function getAllUsers() {
  try {
    const [user1, user2, user3] = await Promise.all([
      fetchUserData(1),
      fetchUserData(2),
      fetchUserData(3)
    ]);
    console.log('All users:', user1, user2, user3);
  } catch (error) {
```

```
    console.error('Error:', error.message);
  }
}
```

5. Explain Event Bubbling and Event Delegation.

Answer:

Event Bubbling: Events propagate from the target element up through its ancestors.

Event Delegation: Using a single event listener on a parent to handle events for multiple children.

Example:

javascript

```
// Event Bubbling
document.getElementById('parent').addEventListener('click', () => {
  console.log('Parent clicked');
});

document.getElementById('child').addEventListener('click', (e) => {
  console.log('Child clicked');
  // e.stopPropagation(); // Stops bubbling
});

// Event Delegation
document.getElementById('todo-list').addEventListener('click', (e)
=> {
  if (e.target.classList.contains('delete-btn')) {
    // Handle delete button click
    e.target.parentElement.remove();
  } else if (e.target.classList.contains('edit-btn')) {
    // Handle edit button click
    const todoText = e.target.previousElementSibling;
    const newText = prompt('Edit todo:', todoText.textContent);
    if (newText) todoText.textContent = newText;
  }
});

// HTML
/*
<ul id="todo-list">
  <li>Task 1 <button class="edit-btn">Edit</button> <button
class="delete-btn">Delete</button></li>
  <li>Task 2 <button class="edit-btn">Edit</button> <button
class="delete-btn">Delete</button></li>
</ul>
```

```
*/
```

6. What is destructuring in JavaScript?

Answer: Destructuring allows extracting values from arrays or properties from objects into distinct variables.

Example:

javascript

```javascript
// Array Destructuring
const colors = ['red', 'green', 'blue', 'yellow'];
const [first, second, ...rest] = colors;
console.log(first); // 'red'
console.log(second); // 'green'
console.log(rest); // ['blue', 'yellow']

// Object Destructuring
const person = {
  name: 'John',
  age: 30,
  city: 'New York',
  country: 'USA'
};

const { name, age, ...address } = person;
console.log(name); // 'John'
console.log(age); // 30
console.log(address); // { city: 'New York', country: 'USA' }

// Renaming variables
const { name: personName, age: personAge } = person;
console.log(personName); // 'John'

// Default values
const { name, age, salary = 50000 } = person;
console.log(salary); // 50000

// Function parameters
function greetPerson({ name, age }) {
  console.log(`Hello ${name}, you are ${age} years old`);
}

greetPerson(person); // 'Hello John, you are 30 years old'
```