1 Function Closures

```
// Function x() creates a closure.
function x() {
    var a = 9;

    // Function y() is defined within x() and has access
       to a.
    function y() {
        console.log(a);
    }

    return y;
}

// z() is a closure that remembers the environment in
   which it was created.
var z = x();
z(); // Outputs 9
```

Explanation: In this code, we define a function 'x()' that creates a closure. The inner function 'y()' is defined within 'x()' and has access to the variable 'a' from its parent function. When we invoke 'z()', it executes 'y()' and prints the value of 'a', which is 9.

2 Nested Function Closures

```
// Function b() creates closures with local variables b
   and i.
function b() {
    let b = 100;
    let i = 900;

    // Function y() is defined within b() and has access
       to b.
    function y() {
        console.log(b);

        // Function k() is defined within y() and has
           access to both b and i.
        function k() {
            console.log(b, i);
        }

```

```
        k();
    }

    y();
}

b(); // Outputs 100 followed by 100 900
```

Explanation: In this code, we define a function ‘b()‘ that creates closures with local variables ‘b‘ and ‘i‘. The inner function ‘y()‘ is defined within ‘b()‘ and has access to ‘b‘. Inside ‘y()‘, another inner function ‘k()‘ is defined, which has access to both ‘b‘ and ‘i‘. When we call ‘b()‘, it invokes ‘y()‘, and subsequently ‘k()‘, resulting in the printed values.

3 Asynchronous JavaScript

```
// setTimeout is used to create asynchronous behavior.
console.log("Learning JavaScript");

setTimeout(() => {
    console.log("Statement 1");
}, 1000); // The 1000 is the duration of the delay in
   milliseconds

console.log("JavaScript is asynchronous");

setTimeout(() => {
    console.log("Statement 2");
}, 0);

console.log("JavaScript is OK OK");

setTimeout(() => {
    console.log("Statement 3");
}, 3000);
```

Explanation: This code demonstrates asynchronous behavior in JavaScript using ‘setTimeout()‘. Messages are logged, and ‘setTimeout()‘ is used to schedule statements to run with different delays. JavaScript continues executing other code while waiting for these timeouts, showcasing its asynchronous nature.

4 Callback Hell (Pyramid of Doom)

```
// Callback example demonstrating Callback Hell (Pyramid
    of Doom).
function operations(){
    dostep1(0,(result)=>{
        dostep2(result,(result2)=>{
            dostep3(result2,(result3)=>{
                console.log(result3);
            });
        });
    });
}

function dostep1(num, callback){
    const result=num+1;
    callback(result);
}

function dostep2(num, callback){
    const result=num+2;
    callback(result);
}

function dostep3(num, callback){
    const result=num+3;
    callback(result);
}

operations();
```

Explanation: This code demonstrates callback functions but is structured in a way that leads to Callback Hell or the Pyramid of Doom. Each function takes a callback to handle the next step, creating nested callbacks that can become hard to read and maintain for complex operations.

5 Promises

```
% Promises example
let success = true;
const promiseObject = new Promise(function(onFulfill,
    onReject) {
    if (success) {
        onFulfill("Successful");
    } else {
        onReject("Rejected");
```

```
    }
});

promiseObject.then(res => console.log(res));
promiseObject.catch(error => console.log(error));
```

Explanation: This code demonstrates the use of Promises to handle asynchronous operations in a structured way. A Promise object is created with two functions, 'onFulfill' and 'onReject', representing success and failure scenarios. Depending on the 'success' variable, the promise either fulfills or rejects. 'then' is used to handle fulfillment, and 'catch' is used to handle rejection.

6 Fetching Data in JavaScript

```
% Fetching Data in JavaScript using the fetch API
fetch('https://jsonplaceholder.typicode.com/todos/1')
    .then(response => response.json())
    .then(json => console.log(json))
```

Explanation: This code demonstrates how to use the 'fetch' API to make an HTTP request to a remote server. The URL "https://jsonplaceholder.typicode.com/todos/1" is used as an example. 'fetch' returns a promise that resolves with the response from the server. The response is converted to JSON using 'response.json()', and the resulting data is logged to the console.