


# Mobile Computing : Lecture-3

Created by	 Ahmed Bashir
@ Email	<a href="mailto:ahmed.bashir@gift.edu.pk">ahmed.bashir@gift.edu.pk</a>

## Mobile Computing : Lecture-3

### Functions in JavaScript

Functions in JavaScript are an essential part of programming in the language. They are blocks of code that perform a specific task, and they are used to perform repetitive tasks. Also, they can be called at any time in the code. JavaScript functions are defined using the `function` keyword, followed by a name and a set of parentheses that may include parameters. The code block is then enclosed in curly braces.

A function is a reusable piece of code that performs a specific task. Functions are used in programming to break down a large program into smaller, more manageable pieces. Functions can be called from anywhere in the program, and they can be called multiple times with different arguments, which makes them very useful.

Here is an example of a simple JavaScript function:

```
function functionName() {  
    console.log("I am a function!");  
}  
  
functionName(); // Output: "I am a function!"
```

You can call or invoke this function using the name of the function followed by parentheses; `functionName();`. For the example above, the function call is `functionName();`. All of the code between curly brackets of functions will be executed each time this function is called.

The function can be called multiple times with different names, and it will print a greeting message for each name.

### Passing Values to Functions

In programming, **parameters** serve as symbolic representations for the values that will be provided as input to a function when it is invoked (called). When a function is defined, it is

typically defined along with one or more parameters. The actual values that we pass when we call the function are known as **arguments**.

Here is an example of a function with two parameters:

```
function test(param1, param2){  
  console.log(param1, param2);  
}
```

We can call the function above like `test("Hello", "Learners")`. We have passed two string arguments, `"Hello"` and `"Learners"`. Inside the function, `param1` will contain the value `"Hello"` and `param2` will contain the value `"Learners"`. Note that you could call the `test` function again with various arguments.

## Returning a Value from a Function

To return a value from a function in JavaScript, you can use the `return` keyword followed by the value you want to return. Here is an example:

```
function add(a, b) {  
  return a + b;  
}  
  
console.log(add(2, 3)); // Output: 5
```

In this example, the `add` function takes two parameters `a` and `b`, and it returns the sum of the two parameters using the `return` keyword. The function can be called multiple times with different parameters, and it will return the sum of the parameters.

## Different ways of writing functions

There are several ways of writing functions in JavaScript. Some of the most common ways of writing functions are:

### Function Declaration

A function declaration is a named function that is defined using the `function` keyword, followed by a name and a set of parentheses that may include parameters. The code block is then enclosed in curly braces.

Here is an example of a function declaration:

```
function add(a, b) {  
  return a + b;  
}  
  
console.log(add(2, 3)); // Output: 5
```

In this example, the function `add` takes two parameters `a` and `b`, and it returns the sum of the two parameters. The function can be called multiple times with different parameters, and it will return the sum of the parameters.

## Function Expression

A function expression is a function that is assigned to a variable. It is defined using the `function` keyword, followed by a set of parentheses that may include parameters. The code block is then enclosed in curly braces, and the entire function is assigned to a variable.

Here is an example of a function expression:

```
const add = function(a, b) {  
  return a + b;  
}  
  
console.log(add(2, 3)); // Output: 5
```

In this example, the function `add` is assigned to a variable `add`. The function takes two parameters `a` and `b`, and it returns the sum of the two parameters. The function can be called multiple times with different parameters, and it will return the sum of the parameters.

## Arrow Function

An arrow function is a shorter way of writing a function expression. It is defined using the `=>` operator, and it does not require the `function` keyword or curly braces.

Here is an example of an arrow function:

```
const add = (a, b) => a + b;  
  
console.log(add(2, 3)); // Output: 5
```

In this example, the function `add` is defined using an arrow function. The function takes two parameters `a` and `b`, and it returns the sum of the two parameters. The function can be called multiple times with different parameters, and it will return the sum of the parameters.

## Hoisting in JavaScript

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their respective scopes before code execution. This means that even if a variable or function is declared later in the code, it can be used before it is declared.

### Variable Hoisting

Variable hoisting refers to the behavior of moving variable declarations to the top of their scope. However, the value of the variable is not moved, only the declaration.

Here's an example:

```
console.log(a); // Output: undefined
var a = 2;
```

In this example, even though the variable `a` is declared after the `console.log` statement, it is still hoisted to the top of its scope. However, the value of `a` is not moved, so the output is `undefined`.

### Function Hoisting

Function hoisting refers to the behavior of moving function declarations to the top of their respective scopes. This means that you can call a function before it is declared in the code.

Here's an example:

```
greet("John"); // Output: "Hello, John!"

function greet(name) {
  console.log(`Hello, ${name}!`);
}
```

In this example, the function `greet` is called before it is declared in the code. However, because of hoisting, the function is moved to the top of its scope, so the output is `Hello, John!`.

It's important to note that function expressions are not hoisted. Only function declarations are hoisted.

```
greet("John"); // Output: Uncaught TypeError: greet is not a function

const greet = function(name) {
```

```
console.log(`Hello, ${name}!`);  
}
```

In this example, the function `greet` is defined using a function expression, so it is not hoisted. Therefore, when it is called before it is defined, a `TypeError` is thrown.

Arrow functions are not hoisted in JavaScript, unlike function declarations. This means that they cannot be used before they are defined in the code.

## Strings and Template Literals

Strings are sequences of characters that represent text. In JavaScript, strings are written inside double quotes (`"..."`) or single quotes (`'...'`). Here are some examples:

```
let greeting = "Hello, world!";  
let message = 'It\'s a beautiful day.';
```

Template literals are a newer way of writing strings in JavaScript. They are enclosed in backticks (``...``), and they allow for more flexibility and readability in string formatting. Here is an example:

```
let name = "John";  
let age = 30;  
let message = `My name is ${name} and I'm ${age} years old.`;
```

In this example, the template literal includes variables using the `${...}` syntax. The resulting string stored in `message` is "My name is John and I'm 30 years old."

Template literals also allow for multi-line strings without the need for escape characters or concatenation. Here is an example:

```
let message = `This is a  
multi-line  
string.`;
```

In this example, the resulting string stored in `message` is "This is a\nmulti-line\nstring.", with each line on a separate line.

Template literals provide a more convenient and flexible way of working with strings in JavaScript, especially when it comes to formatting and readability.

## String Methods

Some of the most commonly used String methods are:

### 1. **charAt()** method

This method returns the character at a specified index in a string. The `charAt()` method takes one parameter, the index of the character to return.

```
const str = "Welcome to JS class";
const char = str.charAt(1); //returns e
console.log("Char at index 1 is: ", char);
```

### 2. **concat()** method

`concat()` method combines two or more strings and returns a new string.

The `concat()` method takes one or more parameters, the strings to be concatenated.

```
const firstName = 'John';
const lastName = 'Doe';
const fullName = firstName.concat(' ', lastName); // Returns 'John Doe'
```

### 3. **indexOf()** method

`indexOf()` method returns the index of the first occurrence of a specified value in a string. The `indexOf()` method takes one parameter, the value to search for.

```
const str = 'Hello, world!';
const index = str.indexOf('w'); // Returns 7
```

### 4. **trim()** method

`trim()` method removes whitespace from both ends of a string. The `trim()` method does not take any parameters.

```
const str = '  Hello, world!  ';
const newStr = str.trim(); // Returns 'Hello, world!'
```

### 5. **toUpperCase()** method

This method converts a string to uppercase. The `toUpperCase()` method does not take any

parameters.

```
const str = 'Hello, world!';  
const newStr = str.toUpperCase(); // Returns 'HELLO, WORLD!'
```

## 6. toLowerCase() method

This method converts a string to lowercase. The `toLowerCase()` method does not take any parameters.

```
const str = 'Hello, world!';  
const newStr = str.toLowerCase(); // Returns 'hello, world!'
```

## 7. replace() method

`replace()` method replaces a specified value in a string with another value.

The `replace()` method takes two parameters, the value to be replaced and the new value.

```
const str = 'Hello, world!';  
const newStr = str.replace('world', 'everyone'); // Returns 'Hello, everyone!'
```

## 8. split() method

`split()` method splits a string into an array of substrings based on a specified separator.

The `split()` method takes one parameter, the separator.

```
const str = 'Hello, world!';  
const arr = str.split(','); // Returns ['Hello', ' world!']
```

## 9. slice() method

This method extracts a part of a string and returns a new string. The `slice()` method takes two parameters, the starting index and the ending index (optional).

```
const str = 'Hello, world!';  
const newStr = str.slice(7, 12); // Returns 'world'
```

# Exception Handling

Exception handling in JavaScript is the process of catching and handling errors that occur during the execution of a program. This is an important part of writing robust and reliable code, as it allows you to gracefully handle unexpected errors and prevent your program from crashing.

In JavaScript, the `try...catch` statement is used to handle exceptions. The `try` block contains the code that might throw an exception, and the `catch` block contains the code that handles the exception if it is thrown.

Here's an example of how to use `try...catch` in JavaScript:

```
try {
  // Code that might throw an exception
  const result = 1 / 0; // Division by zero - throws an exception
} catch (error) {
  // Code that handles the exception
  console.log('An error occurred:', error.message);
}
```

In this example, the `try` block contains a division by zero, which is an operation that throws an exception in JavaScript. The `catch` block catches the exception and logs an error message to the console.

You can also use the `finally` block to execute code that should always run, regardless of whether an exception was thrown or not. Here's an example:

```
try {
  // Code that might throw an exception
  const result = 1 / 0; // Division by zero - throws an exception
} catch (error) {
  // Code that handles the exception
  console.log('An error occurred:', error.message);
} finally {
  // Code that should always run
  console.log('This code will always run, regardless of whether an exception was thrown or not.');
```

## Throwing Exception

In addition to the `try...catch` statement, you can also throw your own exceptions using the `throw` statement. Here's an example:

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero is not allowed.');
```



```
    }  
    return a / b;  
  }  
  
  try {  
    const result = divide(6, 0);  
    console.log(result);  
  } catch (error) {  
    console.log('An error occurred:', error.message);  
  }  
}
```

In this example, the `divide` function throws an exception if the second argument is zero. The `try` block calls the `divide` function with arguments that will cause an exception to be thrown, and the `catch` block handles the exception and logs an error message to the console.