


Mobile Computing : Lecture-4

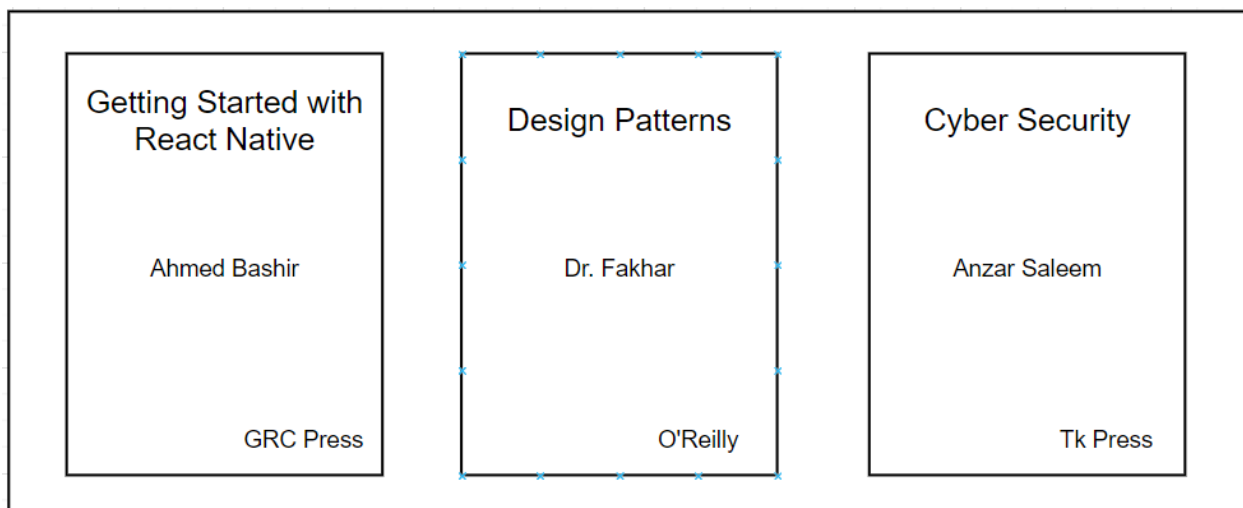
Created by	 Ahmed Bashir
Email	ahmed.bashir@gift.edu.pk

Mobile Computing : Lecture-4

Objects in JavaScript

Objects are reusable components that contain data and functionality. Objects in real life have characteristics that define what they are. For example, a pencil has a length, a diameter, a color, and other characteristics that describe it. A pencil also has things you can do with it, such as write, erase, and sharpen.

Tangible or intangible both type of things can be modeled as a software object. Following is an example of a tangible object **Book**.

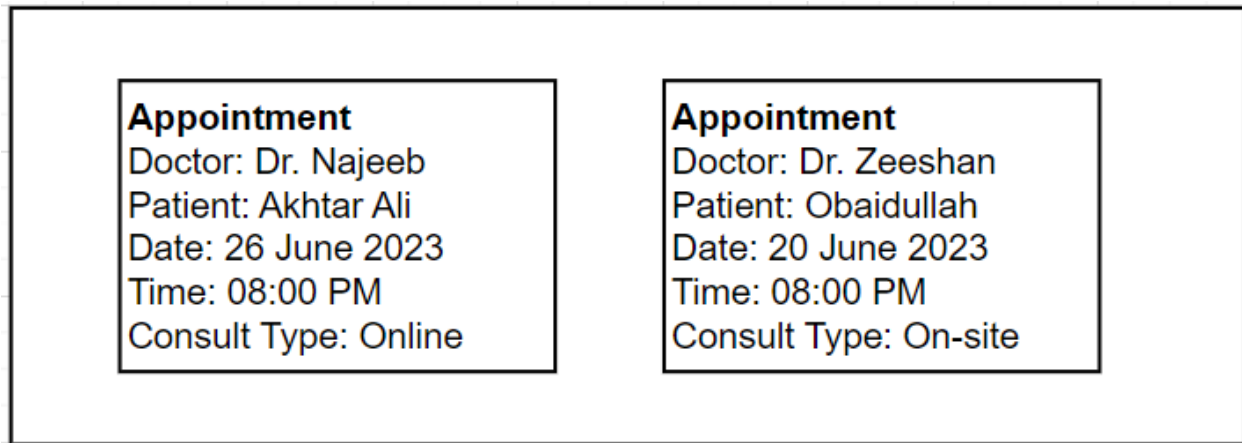


Various Book objects

In the example above, there are different objects of Book. Each book has three properties:

1. Book Title
2. Book Author
3. Book Publisher

Following is an another example which demonstrates an intangible object i.e. **appointment**.



Appointment objects

In the example above, there are two different objects of an Appointment. Each object has several properties like doctor, patient, date, time, and consult type.

JavaScript objects can be used to describe physical objects, but they can also be used to describe abstract ideas.

Objects in JavaScript are similar to **arrays**, except that arrays uses numeric index in an increasing order whereas objects properties(keys). With objects, data can be stored in an structured way.

Creating Objects in JavaScript

There are several ways of creating objects in JavaScript:

1. Using the Object Literal notation
2. Using the new keyword
3. Define a Class
4. Using Object.create()

Making objects using literal notation

To make an object using literal notation, use curly braces containing commaseparated **name:value** pairs. The names become how you access the property, and the value can be any valid JavaScript value, including other objects.

Here's a sample Book object:

```
const book = {
  "title": "You Don't Know JS",
  "author": "Kyle Simpson",
  "publisher": "O'Reilly Media",
  "pages": 325,
  getBookDetails: function(){
    console.log(`${this.title} is written by ${this.author} and published by ${this.publisher}`)
  }
};
```

Access Object Properties

There are two ways to access or modify object properties:

1. Using the dot notation
2. Using the bracket notation similar to arrays

```
//Using the dot notation
const obj = {
  prop1: "val1",
  prop2: "val2"
};

console.log(obj.prop1); // Output: val1
```

The second way to access the properties of an object is bracket notation (`[]`). If the property of the object you are trying to access has a space in its name, you will need to use bracket notation.

However, you can still use bracket notation on object properties without spaces.

```
const exampleObj = {
  "first name": "John",
  "last name": "Smith"
};

console.log(exampleObj["first name"]); // Output: John
```

Modifying Object

Once you've created an object, you can access, modify, and add new properties using one of two methods: dot notation or square brackets notation.

```
//Example: Modifying Object
const person = {}; // An empty object
person.name = "Robert"; // Modified object, added a new property
person.age = 66;

console.log(person);
//Output
{
  "name": "Robert",
  "age": 66
}
```

```
//Example: Modifying an existing object
const book = {
  "title": "Pure React",
  "author": "Dave Ceddia",
};

// Adding a new property
//book.published year // This will not work since the property name contains a space. Use brackets.
book["published year"] = 2020;
book.pages = 360;

console.log(book);
//Output
{
  "title": "Pure React",
  "author": "Dave Ceddia",
  "published year": 2020,
  "pages": 360
}
```

Delete Properties from a JS Object

We can also delete properties from objects like this:

```
delete book.pages;
```

Example:

```
const book = {
  "title": "You Don't Know JS",
  "author": "Kyle Simpson",
  "publisher": "O'Reilly Media",
  "pages": 325,
  getBookDetails: function(){
```

```

        console.log(`${this.title} is written by ${this.author} and published by ${this.publisher}`)
    }
};

delete book.pages;
console.log(book);

//Output
{
  "title": "You Don't Know JS",
  "author": "Kyle Simpson",
  "publisher": "O'Reilly Media"
}

```

Object with Nested Properties

In real scenarios, you encounter objects that contain multiple levels of nested objects. Here's an example:

```

const patient = {
  "name": {
    "firstName": "Shawn",
    "middleName": "",
    "lastName": "Rowden",
  },
  "communicationNumbers": [
    {
      "type": "Primary Phone",
      "number": "(952) 254 9212"
    },
    {
      "type": "Home Telephone",
      "number": "(212) 726 9545"
    },
  ],
  "gender": "Male",
  "isMarried": false
};

```

To get the last name of the patient, you can use the following line:

```
patient.name.lastName
```

Similarly, to get the primary phone:

```
patient.communicationNumbers[0].number // We will refine this since it is hardcoded.
```

Commonly used Object Methods

Method Name	Description	Example
<code>Object.assign()</code>	Copies the values of all enumerable properties from one or more source objects to a target object.	<pre>const target = { a: 1, b: 2 }; const source = { b: 4, c: 5 }; Object.assign(target, source); console.log(target); // Output: { a: 1, b: 4, c: 5 }</pre>
<code>Object.keys()</code>	Returns an array of all the enumerable property names of an object.	<pre>const person = { firstName: 'John', lastName: 'Doe', age: 30 }; const keys = Object.keys(person); console.log(keys); // Output: ["firstName", "lastName", "age"]</pre>
<code>Object.values()</code>	Returns an array of all the enumerable property values of an object.	<pre>const person = { firstName: 'John', lastName: 'Doe', age: 30 }; const values = Object.values(person); console.log(values); // Output: ["John", "Doe", 30]</pre>
<code>Object.entries()</code>	Returns an array of all the enumerable property key-value pairs of an object as an array of arrays.	<pre>const person = { firstName: 'John', lastName: 'Doe', age: 30 }; const entries = Object.entries(person); console.log(entries); // Output: [["firstName", "John"], ["lastName", "Doe"], ["age", 30]]</pre>

Printing out the properties of an Object

```
//document is a global object  
for (const property in document){  
  console.log(`${property}: ${document[property]}`)  
}
```

Making Objects using a Constructor Function

A constructor function is one that can be called with the new keyword to create (or construct) an object.

```
function Cat(name, type){
  this.name = name;
  this.type = type;
}
```

Constructor functions can be used to create multiple objects, and each object created using the Cat() constructor function will have the name and type properties.

The keyword `this` in the above example refers to the context in which a function, such as the constructor function, is running. When you create a new object using the new operator and the Cat() constructor function, `this` refers to the new object.

For example, in the following statement, the this keyword refers to the object named `ourCat` :

```
const outCat = new Cat("Murray", "Domestic short hair");
```

When the Cat() constructor function runs, it does basically the same thing (with some technical differences) as though you had written the following:

```
const ourCat = {};
ourCat.name = 'Murray';
ourCat.type = 'domestic short hair';
```

Destructuring in JavaScript

Destructuring is a feature introduced in ES6 (ECMAScript 2015) that allows developers to extract values from arrays or objects and assign them to variables in a more concise and readable way. Destructuring can make code more readable and easier to maintain by reducing the amount of boilerplate code required to access and manipulate data.

There are two main types of destructuring in JavaScript: **array destructuring** and **object destructuring**.

Array Destructuring

Array destructuring allows developers to extract values from an array and assign them to variables in a single line of code. Here is an example:

```
const myArray = [1, 2, 3];
const [a, b, c] = myArray;
```

```
console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(c); // Output: 3
```

In this example, we declare a new array `myArray` and then use array destructuring to extract its values and assign them to the variables `a`, `b`, and `c`.

Another example:

```
const [a, b] = [1, 2, 3, 4, 5, 6];
console.log(a, b);
```

Swapping Values

Array destructuring can be used to swap the values of two variables without using a temporary variable:

```
let a = 1;
let b = 2;
[a, b] = [b, a];

console.log(a); // Output: 2
console.log(b); // Output: 1
```

In this example, we declare two variables `a` and `b`, and then swap their values using array destructuring.

Ignoring Values

Array destructuring can be used to ignore certain values in an array:

```
const myArray = [1, 2, 3];
const [a, , c] = myArray;

console.log(a); // Output: 1
console.log(c); // Output: 3
```

Another example:

```
const [a, b,,, c] = [1, 2, 3, 4, 5, 6];
console.log(a, b, c);
```

Object Destructuring

Object destructuring works in a similar way to array destructuring, but instead of extracting values from an array, it extracts values from an object. Here is an example:

```
const myObject = { name: 'John', age: 30 };
const { name, age } = myObject;

console.log(name); // Output: "John"
console.log(age); // Output: 30
```

Object destructuring can also be used to assign default values to variables, and to extract values using different variable names:

```
const myObject = { name: 'John' };
const { name: firstName, age = 30 } = myObject;

console.log(firstName); // Output: "John"
console.log(age); // Output: 30
```

Nested Objects

Object destructuring can be used to extract values from nested objects:

```
const myObject = {
  name: 'John',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA'
  }
};

const { name, address: { city } } = myObject;

console.log(name); // Output: "John"
console.log(city); // Output: "Anytown"
```

Renaming Properties

Object destructuring can be used to rename object properties:

```
const myObject = { name: 'John', age: 30 };
const { name: firstName, age: years } = myObject;

console.log(firstName); // Output: "John"
console.log(years); // Output: 30
```

Rest Operator in JavaScript

The rest operator in JavaScript is represented by three dots (`...`) and is used to represent an indefinite number of arguments as an array in a function. The rest operator is part of the ECMAScript 6 (ES6/ES2015) specification, and is commonly used in modern JavaScript development.

The rest operator allows developers to write functions that can accept any number of arguments, without having to specify them in the function declaration. This can make code more flexible and reusable, as well as easier to understand and maintain.

Here's an example of using the rest operator in a function:

```
function sum(...numbers){
  console.log(numbers); //Outputs an array
  let sum = 0;
  for(let i = 0; i < numbers.length; ++i){
    sum = sum + numbers[i];
  }
  return sum;
}

sum(1,2) // 3
sum(100,12,1,3,4,5); // 125
```

The rest operator can also be used with destructuring to extract the remaining elements of an array into a new array:

```
const myArray = [1, 2, 3, 4, 5];
const [a, b, ...rest] = myArray;

console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(rest); // Output: [3, 4, 5]
```

The rest operator in JavaScript can also be used with objects to extract the remaining properties of an object into a new object. This is similar to how the rest operator works with arrays, but with objects, the syntax is slightly different.

Here's an example of using the rest operator with objects:

```
const myObject = { a: 1, b: 2, c: 3, d: 4 };
const { a, b, ...rest } = myObject;

console.log(a); // Output: 1
console.log(b); // Output: 2
console.log(rest); // Output: { c: 3, d: 4 }
```

In this example, we declare a new object `myObject` and then use object destructuring with a rest parameter (`...rest`) to extract the `a` and `b` properties of the object into separate variables, and then assign the remaining properties to the `rest` variable.

Spread Operator in JavaScript

The spread operator in JavaScript is represented by three dots (`...`) and is used to spread the elements of an array or object into a new array or object. The spread operator is part of the ECMAScript 6 (ES6/ES2015) specification, and is commonly used in modern JavaScript development.

The spread operator allows developers to manipulate arrays and objects in a more efficient and concise way. By spreading the elements of an array or object into a new array or object, developers can easily combine, copy, or modify data structures in their code.

Combining Arrays

The spread operator can be used to combine two or more arrays into a new array:

```
const myArray1 = [1, 2, 3];
const myArray2 = [4, 5, 6];
const combinedArray = [...myArray1, ...myArray2];

console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]
```

In this example, we declare two arrays `myArray1` and `myArray2`, and then use the spread operator (`...`) to spread the elements of both arrays into a new array called `combinedArray`.

Copying Arrays

The spread operator can also be used to create a copy of an array:

```
const myArray = [1, 2, 3];
const myCopy = [...myArray];

console.log(myCopy); // Output: [1, 2, 3]
```

In this example, we use the spread operator (`...`) to spread the elements of `myArray` into a new array called `myCopy`, effectively creating a copy of the original array.

Merging Objects

The spread operator can also be used to merge objects:

```
const myObject1 = { a: 1, b: 2 };
const myObject2 = { c: 3, d: 4 };
const mergedObject = { ...myObject1, ...myObject2 };

console.log(mergedObject); // Output: { a: 1, b: 2, c: 3, d: 4 }
```