1. javascript
2. Global execution context
3. Currying

Currying Is a function that take one argument at a tim and return a new function and expecting the next argument.

It is a conversion of function from f(a,b) to f(a)(b)

1. Deep copy vs shallow copy  
     
   call by reference shallow copy mutable object

call by value deep copy immutable object

primitive type

number

boolean

string

null

undefined

primitive type are immutable so there is no concept of deep and shallow copy. Primitive type are shared different memory location.

/\*\*

 \* primitive data type

 \* deep copy

 \*/

let originalValue = 10;

let copiedValue = originalValue;

console.log("original variable", originalValue); //10

console.log("copied variable", copiedValue); //10

copiedValue = 20;

console.log("after change copied variable 10 to 20");

console.log("original variable", originalValue); //10

console.log("copied variable", copiedValue); //20

non primitive

array

object

javascript does shallow copy by default for non primitive type

1. javascript copying one object to another using assignment (=) operator. When we are copying object using assignment operator, if we change anything in copied object it also reflects in the original object, because javascript by default shallow copy.

let originalObj = {

  name: "john",

  age: 20,

};

let copiedObj = originalObj;

console.log("original variable", originalObj);

console.log("copied variable", copiedObj);

copiedObj.age = 40;

console.log("after change copied obj variable age 20 to 40");

console.log("original variable", originalObj);

console.log("copied variable", copiedObj)

1. creating deep copy
2. JSON.stringfy()
3. Object.assign()
4. spread operator (…)
5. JSON.stringfy()

JSON.stringify() methods create a deep copy of a object, when we are copying to another object.

let copiedeepObj = JSON.parse(JSON.stringify(originaldeepObj));

console.log("original variable", originaldeepObj);

console.log("copied variable", copiedeepObj);

copiedeepObj.age = 40;

console.log("after change copied obj variable age 20 to 40");

console.log("original variable", originaldeepObj);

console.log("copied variable", copiedeepObj);

JSON.stringify() method is not enough capable for copying function in a object.

let originassideepObj = {

  name: "john",

  age: 20,

  getName: () => {},

};

/\*\*

 \* JSON.stringify() method is not enough capable for copying function in a object

 \*/

let copiedeepassiObj = JSON.parse(JSON.stringify(originassideepObj));

console.log("original variable", originassideepObj);

console.log("copied variable", copiedeepassiObj)

1. Object.assign()

When we are copying object with functions using Object.assign() it will create a deep copy of that object.

let copieassideepObj = Object.assign({}, originassideepObj);

console.log("original variable", originassideepObj);

console.log("copied variable", copieassideepObj);

copieassideepObj.age = 40;

console.log("after change copied obj variable age 20 to 40");

console.log("original variable", originassideepObj);

console.log("copied variable", copieassideepObj)

Object.assign() methods are creating partial deep copy when object have nested properties. If we change in nested properties value it will also change in original object nested property value.

let origispreaddeepObj = {

  name: "john",

  age: 20,

  getName: () => {},

  address: {

    city: "bangalore",

    state: "karnataka",

  },

};

/\*\*

 \* Object.assign() method create a partial deep copy, if we have nested object and we change in nested value it also reflect in original object

 \*/

let copiedspreaddeepObj = Object.assign({}, origispreaddeepObj);

console.log("original variable", origispreaddeepObj);

console.log("copied variable", copiedspreaddeepObj);

copiedspreaddeepObj.age = 40;

copiedspreaddeepObj.address.city = "magalore"

1. spread operator (…)

when we are copying object using spread operator it also create a partial deep copy. To create a deep copy we need more logic to use spread operator.

let copiedspreaddeepObj = { ...origispreaddeepObj };

console.log("original variable", origispreaddeepObj);

console.log("copied variable", copiedspreaddeepObj);

/\*\*

 \* spread operator also forms a partial deep copy when copying a nested object, to create a deep copy need some more logic to copy object

 \*

 \*/

copiedspreaddeepObj = {

  ...copiedspreaddeepObj,

  age: 41,

  address: {

    ...copiedspreaddeepObj.address,

    city: "shivmoga",

  },

};

console.log("original variable", origispreaddeepObj);

console.log("copied variable", copiedspreaddeepObj)

1. hoisting
2. map, filter and forEach
3. slice and splice
4. arrow function
5. iife
6. callback , pure and higher order function
7. closures
8. promises

javascript promise is a object that encapsulates the result of a asynchronous operation. A promise object has a state that can be one of the following

1. pending
2. fulfilled with value
3. rejected for reason

in the beginning, the state of a promise is pending, indicating that the asynchronous operation is in progress. depending on the result of the asynchronous operation, the state changes to either fulfilled or rejected.

Pending | fulfilled with value

Pending | rejected with error

Creating a promise

To create a promise object, we use the **Promise()** constructor. The promise constructor accepts a callback function that typically performs asynchronous operation. Callback functions accepts two callback functions with name resolve and reject by convention.

If the asynchronous operation completes successfully, the callback will call the resolve() function to change the state of the promise from pending to fulfilled with a value.

In case of error, the callback will call the reject() function to change the state of the promise from pending to rejected with the error reason.

Consuming a promise

1. then
2. catch
3. finally
4. then() method

to get the value of a promise when it’s fulfilled, we call the then() method of the promise object. It accepts two callback function onFulfilled and onRejected.

then() method calls the onFulfilled() with a value, if the promise is fulfilled or the onRejected() with error if the promise is rejected.

promise.then (onFulfilled, onRejected)

1. catch() method

if we want to get error only when the state of the promise is rejected, we can use catch() method of a promise object.it accepts one callback function onRejected.

Internally the catch () method invokes the then (undefined, onRejected).

1. finally() method

when we want to execute the same piece of code whether the promise is fulfilled or rejected.

promise.catch (onRejected)

Promise chaining

Promise.all()

Promise.race()

Promise.any()

Promise.allSettled()

Promise.prototype.finally()

Promise Error handling

Async/await