ES6 - Promises

Promise Syntax

The Syntax related to promise is mentioned below where, **p** is the promise object, **resolve** is the function that should be called when the promise executes successfully and **reject** is the function that should be called when the promise encounters an error.

```
let p = new Promise(function(resolve,reject){
   let workDone = true; // some time consuming work
      if(workDone){
      //invoke resolve function passed

          resolve('success promise completed')
   }
   else{
      reject('ERROR , work could not be completed')
   }
})
```

Example

The example given below shows a function **add_positivenos_async()** which adds two numbers asynchronously. The promise is resolved if positive values are passed. The promise is rejected if negative values are passed.

```
<script>
   function add_positivenos_async(n1, n2) {
      let p = new Promise(function (resolve, reject) {
         if (n1 >= 0 && n2 >= 0) {
            //do some complex time consuming work
            resolve(n1 + n2)
         }
         else
            reject('NOT_Postive_Number_Passed')
         })
         return p;
   }

   add_positivenos_async(10, 20)
      .then(successHandler) // if promise resolved
      .catch(errorHandler);// if promise rejected

   add_positivenos_async(-10, -20)
```

```
      .then(successHandler) // if promise resolved
      .catch(errorHandler);// if promise rejected

   function errorHandler(err) {
      console.log('Handling error', err)
   }
   function successHandler(result) {
      console.log('Handling success', result)
   }

   console.log('end')
</script>
```

The output of the above code will be as mentioned below −

```
end
Handling success 30
Handling error NOT_Postive_Number_Passed
```

Promises Chaining

Promises chaining can be used when we have a sequence of **asynchronous tasks** to be done one after another. Promises are chained when a promise depends on the result of another promise. This is shown in the example below

Example

In the below example, **add_positivenos_async() function** adds two numbers asynchronously and rejects if negative values are passed. The result from the current asynchronous function call is passed as parameter to the subsequent function calls. Note each **then()** method has a return statement.

```
<script>
   function add_positivenos_async(n1, n2) {
      let p = new Promise(function (resolve, reject) {
         if (n1 >= 0 && n2 >= 0) {
            //do some complex time consuming work
            resolve(n1 + n2)
         }
         else
            reject('NOT_Postive_Number_Passed')
      })
      return p;
   }

   add_positivenos_async(10,20)
   .then(function(result){
      console.log("first result",result)
      return add_positivenos_async(result,result)
   }).then(function(result){
```

```
   console.log("second result",result)
      return add_positivenos_async(result,result)
   }).then(function(result){
      console.log("third result",result)
   })

   console.log('end')
</script>
```

The output of the above code will be as stated below −

```
end
first result 30
second result 60
third result 120
```

Some common used methods of the promise object are discussed below in detail −

promise.all()

This method can be useful for aggregating the results of multiple promises.

Syntax

The syntax for the **promise.all()** method is mentioned below, where, **iterable** is an iterable object. E.g. Array.

```
Promise.all(iterable);
```

Example

The example given below executes an array of asynchronous operations **[add_positivenos_async(10,20),add_positivenos_async(30,40),add_positivenos_async(50,60)]**. When all the operations are completed, the promise is fully resolved.

```
<script>
   function add_positivenos_async(n1, n2) {
      let p = new Promise(function (resolve, reject) {
         if (n1 >= 0 && n2 >= 0) {
            //do some complex time consuming work
            resolve(n1 + n2)
         }
         else
            reject('NOT_Postive_Number_Passed')
      })

      return p;
   }
   //Promise.all(iterable)
```

```
Promise.all([add_positivenos_async(10,20),add_positivenos_async(30,40),add_positivenos_async(50,
   .then(function(resolveValue){
      console.log(resolveValue[0])
      console.log(resolveValue[1])
      console.log(resolveValue[2])
      console.log('all add operations done')
   })
   .catch(function(err){
      console.log('Error',err)
   })
   console.log('end')
</script>
```

The output of the above code will be as follows −

```
end
30
70
110
all add operations done
```

promise.race()

This function takes an array of promises and returns the first promise that is settled.

Syntax

The syntax for the **promise.race()** function is mentioned below, where, iterable is an iterable object. E.g. Array.

```
Promise.race(iterable)
```

Example

The example given below takes an array **[add_positivenos_async(10,20),add_positivenos_async(30,40)]** of asynchronous operations.

The promise is resolved whenever any one of the add operation completes. The promise will not wait for other asynchronous operations to complete.

```
<script>
   function add_positivenos_async(n1, n2) {
      let p = new Promise(function (resolve, reject) {
         if (n1 >= 0 && n2 >= 0) {
            //do some complex time consuming work
            resolve(n1 + n2)
```

```
         } else
            reject('NOT_Postive_Number_Passed')
      })

      return p;
   }

   //Promise.race(iterable)
   Promise.race([add_positivenos_async(10,20),add_positivenos_async(30,40)])
   .then(function(resolveValue){
      console.log('one of them is done')
      console.log(resolveValue)
   }).catch(function(err){
      console.log("Error",err)
   })

   console.log('end')
</script>
```

The output of the above code will be as follows −

```
end
one of them is done
30
```

Promises are a clean way to implement async programming in JavaScript (ES6 new feature). Prior to promises, Callbacks were used to implement async programming. Let's begin by understanding what async programming is and its implementation, using Callbacks.

Understanding Callback

A function may be passed as a parameter to another function. This mechanism is termed as a **Callback**. A Callback would be helpful in events.

The following example will help us better understand this concept.

```
<script>
   function notifyAll(fnSms, fnEmail) {
      console.log('starting notification process');
      fnSms();
      fnEmail();
   }
   notifyAll(function() {
      console.log("Sms send ..");
   },
   function() {
      console.log("email send ..");
   });
   console.log("End of script");
```

```
   //executes last or blocked by other methods
</script>
```

In the **notifyAll()** method shown above, the notification happens by sending SMS and by sending an e-mail. Hence, the invoker of the notifyAll method has to pass two functions as parameters. Each function takes up a single responsibility like sending SMS and sending an e-mail.

The following output is displayed on successful execution of the above code.

```
starting notification process
Sms send ..
Email send ..
End of script
```

In the code mentioned above, the function calls are synchronous. It means the UI thread would be waiting to complete the entire notification process. Synchronous calls become blocking calls. Let's understand non-blocking or async calls now.

Understanding AsyncCallback

Consider the above example.

To enable the script, execute an asynchronous or a non-blocking call to notifyAll() method. We shall use the **setTimeout()** method of JavaScript. This method is async by default.

The setTimeout() method takes two parameters −

- A callback function.
- The number of seconds after which the method will be called.

In this case, the notification process has been wrapped with timeout. Hence, it will take a two seconds delay, set by the code. The notifyAll() will be invoked and the main thread goes ahead like executing other methods. Hence, the notification process will not block the main JavaScript thread.

```
<script>
   function notifyAll(fnSms, fnEmail) {
      setTimeout(function() {
         console.log('starting notification process');
         fnSms();
         fnEmail();
      }, 2000);
   }
   notifyAll(function() {
      console.log("Sms send ..");
   },
   function() {
      console.log("email send ..");
   });
   console.log("End of script"); //executes first or not blocked by others
</script>
```

The following output is displayed on successful execution of the above code.

```
End of script
starting notification process
Sms send ..
Email send ..
```

In case of multiple callbacks, the code will look scary.

```
<script>
   setTimeout(function() {
      console.log("one");
      setTimeout(function() {
         console.log("two");
         setTimeout(function() {
            console.log("three");
         }, 1000);
      }, 1000);
   }, 1000);
</script>
```

ES6 comes to your rescue by introducing the concept of promises. Promises are "Continuation events" and they help you execute the multiple async operations together in a much cleaner code style.

Example

Let's understand this with an example. Following is the syntax for the same.

```
var promise = new Promise(function(resolve , reject) {
   // do a thing, possibly async , then..
   if(/*everthing turned out fine */)    resolve("stuff worked");
   else
   reject(Error("It broke"));
});
return promise;
// Give this to someone
```

The first step towards implementing the promises is to create a method which will use the promise. Let’s say in this example, the **getSum()** method is asynchronous i.e., its operation should not block other methods’ execution. As soon as this operation completes, it will later notify the caller.

The following example (Step 1) declares a Promise object ‘var promise’. The Promise Constructor takes to the functions first for the successful completion of the work and another in case an error happens.

The promise returns the result of the calculation by using the resolve callback and passing in the result, i.e., n1+n2

Step 1 − resolve(n1 + n2);

If the getSum() encounters an error or an unexpected condition, it will invoke the reject callback method in the Promise and pass the error information to the caller.

Step 2 − reject(Error("Negatives not supported"));

The method implementation is given in the following code (STEP 1).

```
function getSum(n1, n2) {
   varisAnyNegative = function() {
      return n1 < 0 || n2 < 0;
   }
   var promise = new Promise(function(resolve, reject) {
      if (isAnyNegative()) {
         reject(Error("Negatives not supported"));
      }
      resolve(n1 + n2)
   });
   return promise;
}
```

The second step details the implementation of the caller (STEP 2).

The caller should use the ‘then’ method, which takes two callback methods - first for success and second for failure. Each method takes one parameter, as shown in the following code.

```
getSum(5, 6)
.then(function (result) {
   console.log(result);
},
function (error) {
   console.log(error);
});
```

The following output is displayed on successful execution of the above code.

```
11
```

Since the return type of the getSum() is a Promise, we can actually have multiple ‘then’ statements. The first 'then' will have a return statement.

```
getSum(5, 6)
.then(function(result) {
   console.log(result);
   returngetSum(10, 20);
   // this returns another promise
},
function(error) {
   console.log(error);
})
.then(function(result) {
   console.log(result);
```

```
},
function(error) {
   console.log(error);
});
```

The following output is displayed on successful execution of the above code.

```
11
30
```

The following example issues three then() calls with getSum() method.

```
<script>
   function getSum(n1, n2) {
      varisAnyNegative = function() {
         return n1 < 0 || n2 < 0;
      }
      var promise = new Promise(function(resolve, reject) {
         if (isAnyNegative()) {
            reject(Error("Negatives not supported"));
         }
         resolve(n1 + n2);
      });
      return promise;
   }
   getSum(5, 6)
   .then(function(result) {
      console.log(result);
      returngetSum(10, 20);
      //this returns another Promise
   },
   function(error) {
      console.log(error);
   })
   .then(function(result) {
      console.log(result);
      returngetSum(30, 40);
      //this returns another Promise
   },
   function(error) {
      console.log(error);
   })
   .then(function(result) {
      console.log(result);
   },
   function(error) {
      console.log(error);
   });
   console.log("End of script ");
</script>
```

The following output is displayed on successful execution of the above code.

The program displays ‘end of script’ first and then results from calling getSum() method, one by one.

```
End of script
11
30
70
```

This shows getSum() is called in async style or non-blocking style. Promise gives a nice and clean way to deal with the Callbacks.