ES6 Map

ES6 is a series of new features that are added to the JavaScript. Prior to ES6, when we require the mapping of keys and values, we often use an object. It is because the object allows us to map a key to the value of any type.

ES6 provides us a new collection type called **Map**, which holds the key-value pairs in which values of any type can be used as either keys or values. A Map object always remembers the actual insertion order of the keys. Keys and values in a Map object may be primitive or objects. It returns the new or empty Map.

Maps are ordered, so they traverse the elements in their insertion order.

Syntax

For creating a new Map, we can use the following syntax:

```
1. var map = new Map([iterable]);
```

The **Map ()** accepts an optional iterable object, whose elements are in the key-value pairs.

Map Properties

S.no.	Properties	Description
1.	**Map.prototype.size**	This property returns the number of key-value pair

Let us understand the above property of Map object in brief.

Map.prototype.size

It returns the number of elements in the Map object.

Syntax

```
1. Map.size
```

Example

```
1. var map = new Map();
2.     map.set('John', 'author');
3.     map.set('arry', 'publisher');
```

```
    map.set('Mary', 'subscriber');
    map.set('James', 'Distributor');

console.log(map.size);
```

Output

```
4
```

Map Methods

The Map object includes several methods, which are tabulated as follows:

S.no.	Methods	Description
1.	Map.prototype.clear()	It removes all the keys and values pairs fro
2.	Map.prototype.delete(key)	It is used to delete an entry.
3.	Map.prototype.has(value)	It checks whether or not the corresponding
4.	Map.prototype.entries()	It is used to return a new iterator object tha element in the Map object in insertion order
5.	Map.prototype.forEach(callbackFn[, thisArg])	It executes the **callback** function once, whi
6.	Map.prototype.keys()	It returns an iterator for all keys in the Map
7.	Map.prototype.values()	It returns an iterator for every value in the

Weak Maps

Weak Maps are almost similar to normal Maps except that the keys in weak maps must be objects. It stores each element as a key-value pair where keys are weakly referenced. Here, the keys are objects, and the values are arbitrary values. A Weak Map object only allows the keys of an object type. If there is no reference to a key object, then they are targeted to garbage collection. In weak Map, the keys are not enumerable. So, there is no method to get the list of keys.

A weak map object iterates its elements in the insertion order. It only includes **delete(key), get(key), has(key)** and **set(key, value)** method.

Let us try to understand the illustration of the weak Map.

Example

```
'use strict'
let wp = new WeakMap();
let obj = {};
console.log(wp.set(obj,"Welcome to javaTpoint"));
console.log(wp.has(obj));
```

Output

```
WeakMap { <items unknown> }
true
```

The for...of loop and Weak Maps

The for...of loop is used to perform an iteration over keys, values of the Map object. The following example will illustrate the traversing of the Map object by using a for...of loop.

Example

```
'use strict'
var colors = new Map([
   ['1', 'Red'],
   ['2', 'Green'],
   ['3', 'Yellow'],
   ['4', 'Violet']
]);

for (let col of colors.values()) {
    console.log(col);
}

console.log(" ")

for(let col of colors.entries())
console.log(`${col[0]}: ${col[1]}`);
```

Output

```
Red
Green
```

```
Yellow
Violet

1: Red
2: Green
3: Yellow
4: Violet
```

Iterator and Map

An iterator is an object, which defines the sequence and a return value upon its termination. It allows accessing a collection of objects one at a time. Set and Map both include the methods that return an iterator.

Iterators are the objects with the **next()** method. When the **next()** method gets invoked, the iterator returns an object along with
the **"value"** and **"done"** properties.

Let us try to understand some of the implementations of iterator along with the Map object.

Example-1

```
1. 'use strict'
2. var colors = new Map([
3.    ['1', 'Red'],
4.    ['2', 'Green'],
5.    ['3', 'Yellow'],
6.    ['4', 'Violet']
7. ]);
8.
9. var itr = colors.values();
10. console.log(itr.next());
11. console.log(itr.next());
12. console.log(itr.next());
```

Output

```
{ value: 'Red', done: false }
{ value: 'Green', done: false }
{ value: 'Yellow', done: false }
```

Example-2

```
1. 'use strict'
2. var colors = new Map([
```

```
    ['1', 'Red'],
    ['2', 'Green'],
    ['3', 'Yellow'],
    ['4', 'Violet']
]);
var itr = colors.entries();
console.log(itr.next());
console.log(itr.next());
console.log(itr.next());
```

Output

```
{ value: [ '1', 'Red' ], done: false }
{ value: [ '2', 'Green' ], done: false }
{ value: [ '3', 'Yellow' ], done: false }
```

Example-3

```
'use strict'
var colors = new Map([
    ['1', 'Red'],
    ['2', 'Green'],
    ['3', 'Yellow'],
    ['4', 'Violet']
]);

var itr = colors.keys();
console.log(itr.next());
console.log(itr.next());
console.log(itr.next());
```

Output

```
{ value: '1', done: false }
{ value: '2', done: false }
{ value: '3', done: false }
```

ES6 Set

A set is a data structure that allows you to create a collection of unique values. Sets are the collections that deal with single objects or single values.

Set is the collection of values similar to arrays, but it does not contain any duplicates. It allows us to store unique values. It supports both primitive values and object references.

As similar to maps, sets are also ordered, i.e., the elements in sets are iterated in their insertion order. It returns the set object.

Syntax

```
1. var s = new Set("val1","val2","val3");
```

Let us understand the concept of **set** by using the following example:

Example

```
1. let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
2. console.log(colors);
```

All the elements of a Set must be unique. Therefore the set **colors** in the above example only contain four distinct elements. We will get the following output after the successful execution of the above code.

Output

```
Set { 'Green', 'Red', 'Orange', 'Yellow' }
```

Let us see the properties and methods of the Set.

Set Properties

S.no.	Properties	Description
1.	**Set.size**	This property returns the number of values in the set objec

Set.size

This property of the Set object returns the value that represents the number of elements in the Set object.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
console.log(colors.size);
console.log(colors);
```

Output

```
4
Set { 'Green', 'Red', 'Orange', 'Yellow' }
```

Set Methods

The set object includes several methods, which are tabulated as follows:

S.no.	Methods	Description
1.	**Set.prototype.add(value)**	It appends a new element to the give
2.	**Set.prototype.clear()**	It removes all the elements from the
3.	**Set.prototype.delete(value)**	It removes the element which is asso
4.	**Set.prototype.entries()**	It returns a new iterator object, which object in insertion order.
5.	**Set.prototype.forEach(callbackFn[, thisArg])**	It executes the callback function once
6.	**Set.prototype.has(value)**	This method returns true when the pa
7.	**Set.prototype.values()**	It returns the new iterator object, whi Set, in insertion order.

Now, we are going to understand the above methods of the Set object in detail.

Set.prototype.add(value)

This method is used to append a new element with the existing value to the Set object.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
console.log(colors.size);
console.log(colors);
```

Output

```
7
Set { 'Green', 'Red', 'Orange', 'Yellow', 'Violet', 'Indigo', 'Blue' }
```

Set.prototype.clear()

It clears all the objects from the sets.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
colors.clear()
console.log(colors.size);
```

Output

```
0
```

Set.prototype.delete(value)

This method is used to remove the corresponding passed value from the set object.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
colors.delete('Violet');
console.log(colors.size);
```

8. console.log(colors);

Output

```
6
Set { 'Green', 'Red', 'Orange', 'Yellow', 'Indigo', 'Blue' }
```

Set.prototype.entries()

It returns the object of a new set iterator. It contains an array of values for each element. It maintains the insertion order.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
var itr = colors.entries();
for(i=0;i<colors.size;i++)  {
    console.log(itr.next().value);
}
```

Output

```
[ 'Green', 'Green' ]
[ 'Red', 'Red' ]
[ 'Orange', 'Orange' ]
[ 'Yellow', 'Yellow' ]
[ 'Violet', 'Violet' ]
[ 'Indigo', 'Indigo' ]
[ 'Blue', 'Blue' ]
```

Set.prototype.forEach(callbackFn[, thisArg])

It executes the specified callback function once for each Map entry.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
function details(values){
```

```
    console.log(values);
}
colors.forEach(details);
```

Output

```
Green
Red
Orange
Yellow
Violet
Indigo
Blue
```

Set.prototype.has(value)

It returns the Boolean value that indicates whether the element, along with the corresponding value, exists in a Set object or not.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');
colors.add('Indigo');
colors.add('Blue');
colors.add('Violet');
console.log(colors.has('Indigo'));
console.log(colors.has('Violet'));
console.log(colors.has('Cyan'));
```

Output

```
true
true
false
```

Set.prototype.values()

It returns a new iterator object that includes the values for each element in the Set object in the insertion order.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
colors.add('Violet');

```

```
var val = colors.values();
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
console.log(val.next().value);
```

Output

```
Green
Red
Orange
Yellow
Violet
```

Weak Set

It is used to store the collection of objects. It is similar to the Set object, so it also cannot store duplicate values. Similar to weak maps, weak sets cannot be iterated. Weak sets can contain only objects which may be garbage collected.

Weak set only includes **add(value), delete(value)** and **has(value)** methods of Set object.

Example

```
'use strict'
  let ws = new WeakSet();
  let obj = {msg:"Welcome Back!"};
  ws.add(obj);
  console.log(ws.has(obj));
  ws.delete(obj);
  console.log(ws.has(obj));
```

Output

```
true
false
```

Iterators

An iterator is an object which defines the sequence and a return value upon its termination. It allows accessing a collection of objects one at a time. Set and Map both include the methods that return an iterator.

Iterators are the objects with the **next()** method. When the **next()** method gets invoked, the iterator returns an object along with the **'value'** and **'done'** properties.

The **'done'** is a Boolean which returns true after reading all of the elements in the collection. Otherwise, it returns false.

Let's understand the implementations of iterator along with the **Set** object.

Example

```
let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
var itr = colors.keys();
var itr1 = colors.entries();
var itr2 = colors.values();
console.log(itr.next());
console.log(itr1.next());
console.log(itr2.next());
```

Output

```
{ value: 'Green', done: false }
{ value: [ 'Green', 'Green' ], done: false }
{ value: 'Green', done: false }
```

ES6 Promises

A Promise represents something that is eventually fulfilled. A Promise can either be rejected or resolved based on the operation outcome.

ES6 Promise is the easiest way to work with asynchronous programming in JavaScript. Asynchronous programming includes the running of processes individually from the main thread and notifies the main thread when it gets complete. Prior to the Promises, **Callbacks** were used to perform asynchronous programming.

Callback

A Callback is a way to handle the function execution after the completion of the execution of another function.

A Callback would be helpful in working with events. In Callback, a function can be passed as a parameter to another function.

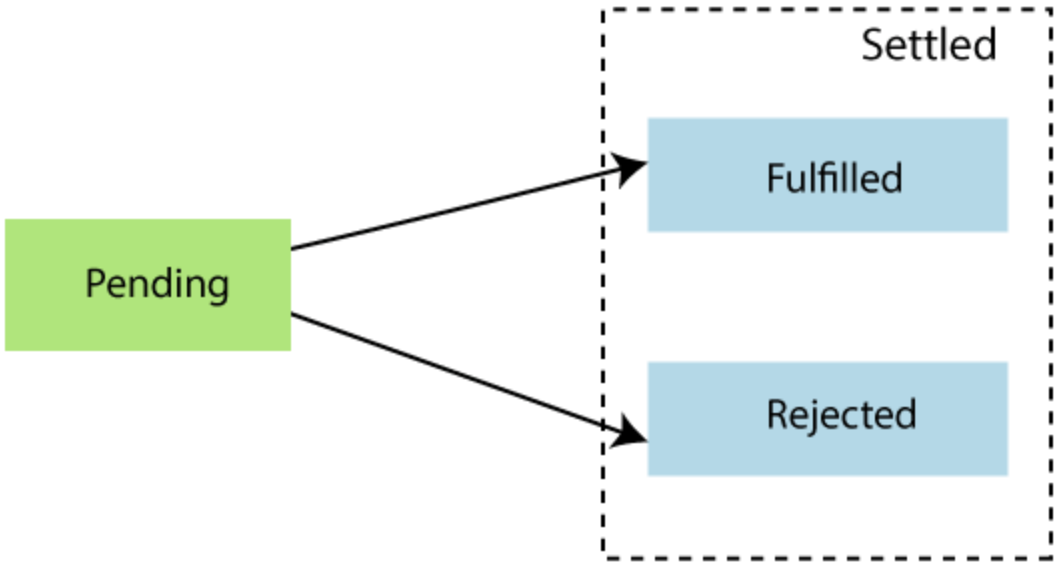
Why Promise required?

A Callback is a great way when dealing with basic cases like minimal asynchronous operations. But when you are developing a web application that has a lot of code, then working with Callback will be messy. This excessive Callback nesting is often referred to as **Callback hell**.

To deal with such cases, we have to use **Promises** instead of **Callbacks**.

How Does Promise work?

The Promise represents the completion of an **asynchronous operation**. It returns a single value based on the operation being **rejected** or **resolved**. There are mainly three stages of the Promise, which are shown below:



Pending - It is the initial state of each Promise. It represents that the result has not been computed yet.

Fulfilled - It means that the operation has completed.

Rejected - It represents a failure that occurs during computation.

Once a Promise is fulfilled or rejected, it will be immutable. The **Promise()** constructor takes two arguments that are **rejected** function and a **resolve** function. Based on the asynchronous operation, it returns either the first argument or second argument.

Creating a Promise

In JavaScript, we can create a Promise by using the **Promise()** constructor.

Syntax

```
const Promise = new Promise((resolve,reject) => {....});
```

Example

```
let Promise = new Promise((resolve, reject)=>{
    let a = 3;
    if(a==3){
        resolve('Success');
    }
    else{
```

```
7.        reject('Failed');
8.    }
9. })
10.       Promise.then((message)=>{
11.          console.log("It is then block. The message is: "+ message)
12.       }).catch((message)=>{
13.       console.log("It is Catch block. The message is: "+ message)
14.       })
```

Output

```
It is then block. The message is: Success
```

Promise Methods

The Promise methods are used to handle the rejection or resolution of the **Promise** object. Let's understand the brief description of Promise methods.

.then()

This method invokes when a Promise is either fulfilled or rejected. This method can be chained for handling the rejection or fulfillment of the Promise. It takes two functional arguments for **resolved** and **rejected**. The first one gets invoked when the Promise is fulfilled, and the second one (which is optional) gets invoked when the Promise is rejected.

Let's understand with the following example how to handle the Promise rejection and resolution by using **.then() method**.

Example

```
1. let success = (a) => {
2.    console.log(a + " it worked!")
3.  }
4.
5.  let error = (a) => {
6.    console.log(a + " it failed!")
7.  }
8.
9.  const Promise = num => {
10.          return new Promise((resolve,reject) => {
11.           if((num%2)==0){
12.             resolve("Success!")
```

```
13.           }
14.           reject("Failure!")
15.         })
16.       }
17.
18.       Promise(100).then(success, error)
19.       Promise(21).then(success,error)
```

Output

```
Success! it worked!
Failure! it failed!
```

.catch()

It is a great way to handle failures and rejections. It takes only one functional argument for handling the errors.

Let's understand with the following example how to handle the Promise rejection and failure by using **.catch() method**.

Example

```
1. const Promise = num => {
2.    return new Promise((resolve,reject) => {
3.      if(num > 0){
4.        resolve("Success!")
5.      }
6.      reject("Failure!")
7.    })
8.  }
9.
10.       Promise(20).then(res => {
11.         throw new Error();
12.         console.log(res + " success!")
13.       }).catch(error => {
14.         console.log(error + " oh no, it failed!")
15.       })
```

Output

```
Error oh no, it failed!
```

.resolve()

It returns a new Promise object, which is resolved with the given value. If the value has a **.then()** method, then the returned Promise will follow that **.then()** method adopts its eventual state; otherwise, the returned Promise will be fulfilled with value.

```
Promise.resolve('Success').then(function(val) {
    console.log(val);
  }, function(val) {
  });
```

.reject()

It returns a rejected **Promise** object with the given value.

Example

```
function resolved(result) {
    console.log('Resolved');
  }

  function rejected(result) {
    console.error(result);
  }

  Promise.reject(new Error('fail')).then(resolved, rejected);
```

Output

```
Error: fail
```

.all()

It takes an array of Promises as an argument. This method returns a **resolved Promise** that fulfills when all of the Promises which are passed as an iterable have been fulfilled.

Example

```
const PromiseA = Promise.resolve('Hello');
const PromiseB = 'World';
const PromiseC = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 1000);
```

```
});

Promise.all([PromiseA, PromiseB, PromiseC]).then(function(values) {
  console.log(values);
});
```

Output

```
[ 'Hello', 'World', 1000 ]
```

.race()

This method is used to return a resolved Promise based on the first referenced Promise that resolves.

Example

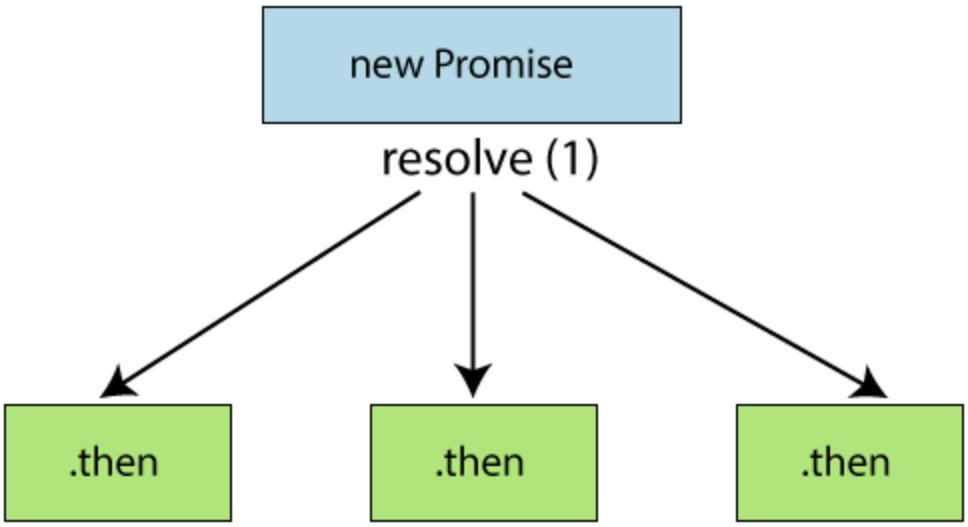
```
const Promise1 = new Promise((resolve,reject) => {
    setTimeout(resolve("Promise 1 is first"),1000)
})

const Promise2= new Promise((resolve,reject) =>{
    setTimeout(resolve("Promise 2 is first"),2000)
})

Promise.race([Promise1,Promise2]).then(result => {
            console.log(result);
      })
```

Output

```
Promise 1 is first
```

ES6 Promise Chaining

Promise chaining allows us to control the flow of JavaScript asynchronous operations. By using Promise chaining, we can use the returned value of a Promise as the input to another asynchronous operation.



Sometimes, it is desirable to chain Promises together. For example, suppose we have several asynchronous operations to be performed. When one operation gives us data, we will start doing another operation on that piece of data and so on.

Promise chaining is helpful when we have multiple interdependent asynchronous functions, and each of these functions should run one after another.

Let us try to understand the concept of Promise chaining by using the following example:

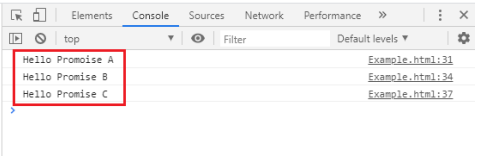
Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Document</title>
</head>
      <body>
         <script>
            const PromiseA = () =>{
               return new Promise((resolve,reject)=>{
                  resolve("Hello Promoise A");
               });
            }
```

```
17.
18.             const PromiseB = () =>{
19.                 return new Promise((resolve,reject)=>{
20.                     resolve("Hello Promise B");
21.                 });
22.             }
23.
24.
25.             const PromiseC = () =>{
26.                 return new Promise((resolve,reject)=>{
27.                     resolve("Hello Promise C");
28.                 });
29.             }
30.             PromiseA().then((A)=>{
31.                     console.log(A);
32.                       return PromiseB();
33.                   }).then((B)=>{
34.                       console.log(B);
35.                       return PromiseC();
36.                   }).then((C)=>{
37.                       console.log(C);
38.                   });
39.         </script>
40.     </body>
41.     </html>
```

Execute the above code in the browser and open the terminal by using **ctrl+shift+I**. After the successful execution of the above code, we will get the following output.

Output



ES6 Animation

Animations in JavaScript can handle the things that CSS can't. Several elements in JavaScript help to create a complex animation such as **Fade effect, Fireworks, Roll-in or Roll-out,** etc. By using JavaScript, we can move the DOM elements such as **</img>, ** or any other HTML element.

There are two ways to perform animation in JavaScript, which are as follows:

- **Using the setInterval() method:** It takes two arguments that are a function and an integer. This method terminates by using the **clearInterval()** method.
- **Using the requestAnimationFrame() method:** This method runs the function when the screen is ready to accept the next repaint. It takes a single argument function. The animated effect occurs when we recursively call this method. The expected animation is created frame by frame, and every frame is called when the browser is ready for it.

Let us try to elaborate on these methods.

setInterval() method

It is the traditional method of JavaScript for producing animation effects. It relies on the repeated execution of code for making changes in the element frame by frame.

```
anime = setInterval(show, t);
//It calls the function show after every t milliseconds
clearInterval(anime);
//It terminates above process
```

If the function **show** produces a change in element style, then **setInterval(show, t) method** can be used for producing gradual changes in the style of an element after every time interval. When the time interval is small, then the animation looks continuous.

requestAnimationFrame() method

This method is easy to set up and hard to cancel. It is optimized to perform smooth animation. The loops in it are needed to be created manually, and also the timing needs to be set up manually. This method is not made to be used at specific intervals.

This method is designed to loop at **60fps (frames per second)** to perform smooth animation. It won't run in the background, and it is also energy efficient.

Now, let's see some of the demonstrations of JavaScript Animation.

Example-1

In this example, we are implementing a simple animation by using the properties of the DOM object and functions of JavaScript. We use the JavaScript function **getElementById()** for getting the DOM object and then assign that object into a global variable.

Let's understand the simple animation by using the following example.

Example

```
<html>
  <head>
    <script type = "text/javascript">

        var img = null;
        function init(){
          img = document.getElementById('myimg');
          img.style.position = 'relative';
          img.style.left = '50px';
        }
        function moveRight(){
          img.style.left = parseInt(
          img.style.left) + 100 + 'px';
        }
        window.onload = init;

    </script>
  </head>

  <body>
    <form>
      <img id = "myimg" src = "train1.png" />
      <center>
        <p>Click the below button to move the image right</p>
      <input type = "button" value = "Click Me" onclick = "moveRight();" />
    </center>
    </form>
  </body>

</html>
```

Example-2

Let's understand another example of JavaScript animation.

In this animation, we will use the **setTimeout()** function of JavaScript. It is obvious that if we are using a **setTimeout()** function, then to clear the timer, we have to set **clearTimeout()** function of JavaScript manually.

In the above example, we saw how the image moves towards right on every click. Let us try to automate this process with the **setTimeout()** function of JavaScript.

```
<html>
  <head>
    <title>JavaScript Animation</title>
    <script type = "text/javascript">
        var anime ;
        function init(){
          img = document.getElementById('myimg');
          img.style.position = 'relative';
          img.style.left = '0px';
        }
        function towardRight(){
          img.style.left = parseInt(img.style.left) + 10 + 'px';
          anime = setTimeout(towardRight,50);
        }
        function stop() {
          clearTimeout(anime);
         img.style.left = '0px';
        }
        window.onload = init;
    </script>
  </head>

  <body>
    <form>
      <img id = "myimg" src = "train1.png" />
      <center>
      <h2 style="color:darkblue;">Click the following buttons to handle animation</h2>
      <input type="button" value="Start" onclick = "towardRight();" />
      <input type = "button" value="Stop" onclick = "stop();" />
```

```
      <center>
    </form>
  </body>
</html>
```

Image Rollover with a mouse event

Let's understand another example of animation in which there is an image rollover by a mouse event. When the mouse moves over the image, the HTTP image will change to the second one from the first image. But when the mouse gets moved away from the image, then the original image will be displayed.

Example

The **onMouseOver** event handler triggers when the user will move the mouse onto the link, and the **onMouseOut** event handler gets trigger when the user will move the mouse away from the link.

```
<html>

  <head>

    <script type = "text/javascript">

        if(document.images) {
          var img1 = new Image();
          img1.src = "cat.jpg";
          var img2 = new Image();
          img2.src = "tiger.jpg";
        }

    </script>
  </head>

  <body>
  <center>

    <a href = "#" onMouseOver = "document.myImg.src = img2.src;"
      onMouseOut = "document.myImg.src = img1.src;">
      <img name = "myImg" src = "cat.jpg" />
    </a><br><br><br>
    <h1>Move your mouse over the image to see the result</h1>
```

```
25.    </center>
26.    </body>
27. </html>
```