**JAVASCRIPT**

1)**FUNCTIONS**:

* A function is a reusable component of a JavaScript code.
* They belong to type of Object and bring about modularity in code writing.
* Every Function has as function declaration and a function body.
* Functions can be created on fly and hence can be unnamed, called as Anonymous Functions.
* Functions generally perform computations by accepting or without accepting parameters and may/may not return the result thus computed.
* Functions can be made concise starting from the ES6 release by writing anonymous functions using concept of Arrow Functions.

**PROBLEM STATEMENT**

String is provided as input with/without spaces, convert the string as follows:

Input: San Jose

Output: JSaenos, (ACII order)if the length of string in greater than >4.

**CODE:**

function checkLength(str){

if((str.length)>4){

return true;

}

else{

return false;

}

}

var sortASCII=(str)=>{

if(checkLength(str)){

return str.split("").sort().join("");

}

else{

return str;

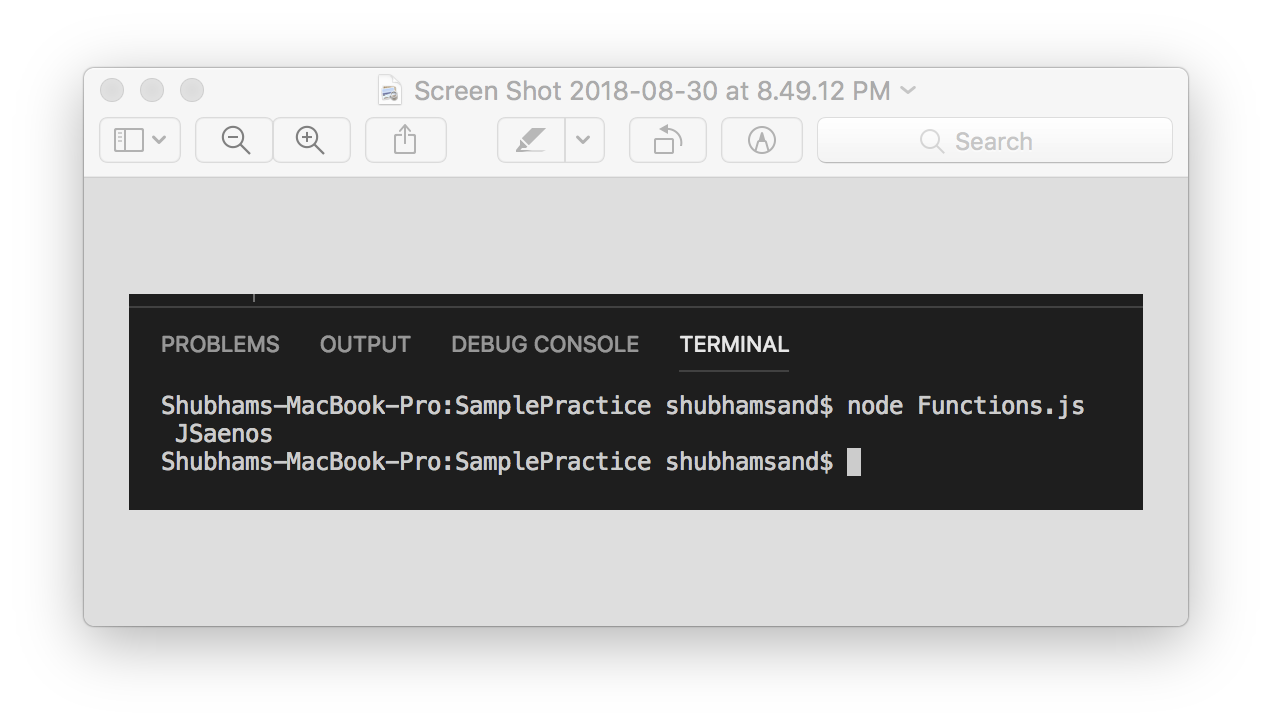
}

};

var inOrder=sortASCII("San Jose");

console.log(inOrder);

**OUTPUT**



**2)EVENTS**

* Users interaction with the web page on browser is managed with the help of events making the graphical user interface more interactive.
* Events are part of the window object that is the root object of the web page and HTML tags can be assigned with various events like: onload, onclick, onmouseout, onmouseover etc. to run JavaScript code.

**PROBLEM STATEMENT:**

Change the color of the button text from black to red on click of the button.

**CODE:**

**HTML:**

<html>

<head></head>

<body>

<button id="demo" style="height: 105px; width: 300px" onclick="changeColor()">Change Color</button>

<script src="events.js"> </script>

</body>

</html>

**JAVASCRIPT:**

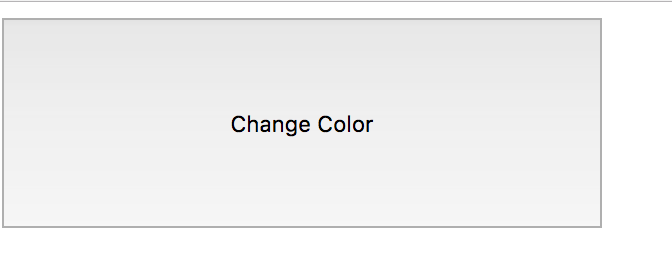
function changeColor(){

document.getElementById("demo").style.color="red";

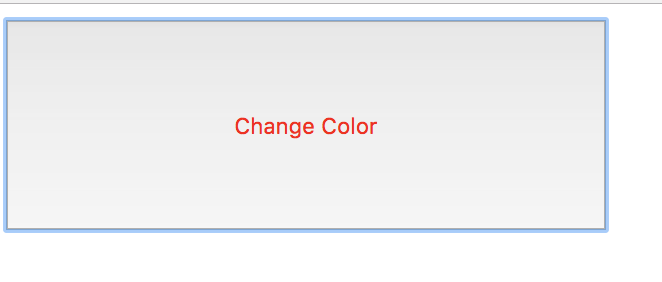
}

**OUTPUT:**

Before Click



After Click



**3)ARRAYS**

* Arrays are used to store multiple similar type of values.
* Arrays are of type object and the variable to which it is assigned points to the memory location where array object is stored.
* Arrays are declared as var arr\_name=[]; and the elements of an array can be accessed with the help of indexes starting from 0 .
* Arrays support various methods that can operate on it like pop(),push(),splice(),shift(),unshift() etc.

**PROBLEM STAEMENT:**

Accept two strings and concatenate it with \* replacing every space between them. And first word of 1st string should be removed from the final string.

Input= 1st string “Apple is sweet”

2nd string “and red in colour”

Oputput=is\*sweet\*and\*red\*in\*colour

**CODE:**

function mergeWithPattern(str1,str2){

var str3=str1+" "+str2;

let arr=str3.split(" ");

arr.shift();

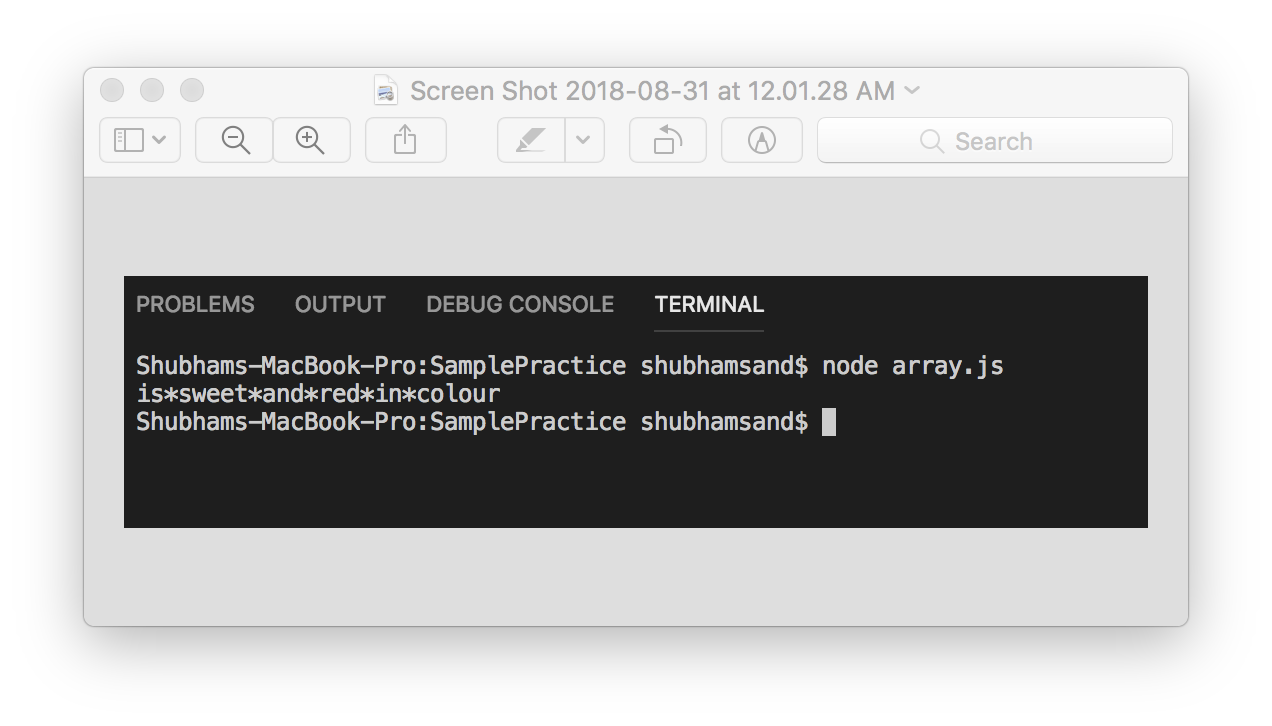
var patternString=arr.join("\*");

return patternString;

}

console.log(mergeWithPattern("Apple is sweet","and red in colour"));

**OUTPUT:**



**4)REGULAR EXPRESSIONS**

* Regular Expressions are used primarily for following reasons:

1)Searching a string for a particular pattern.

2)Replacing a string with a particular pattern

* Regular expressions belong to the RegExp() object.
* Regular expressions in literal form are build using the /…/syntax.
* It supports various flags like the i=case insensitive flag, g=global flag etc.

**PROBLEM STATEMENT:**

Check if the vehicle number is valid or not.

Valid vehicle no- MH 20 CX 9098

**CODE:**

function checkVehicleNumber(str){

var res= /[A-Z]{2} \d{2} [A-Z]{2} \d{4}$/.test(str);

return res;

}

if(checkVehicleNumber("MH 20 CX 9098dfdf")){

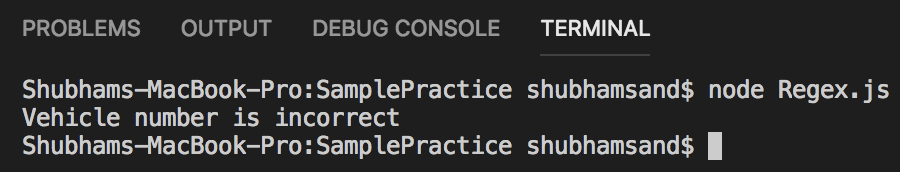
console.log("Vehicle number is correct");

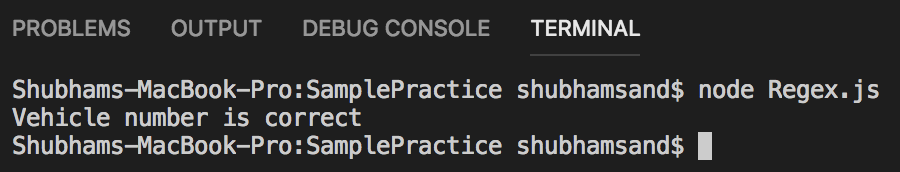
}else{

console.log("Vehicle number is incorrect");

}

**OUTPUT:**





**5)STRICT MODE**

* When strict mode is turned on using “use strict” instruction its puts restriction on relatively tricky parts of JavaScript code which also facilitates better execution by the JavaScript interpreter.
* There are some errors covered up by normal JavaScript code but when we use strict mode then errors are reported.
* Strict mode can be applied to the entire code or also to individual functions.
* Strict mode also prohibits use of reserve keywords as identifiers that would be used in future versions.

**PROBLEM STATEMENT:**

Check if a particular student is valid to get a promotional discount.

Conditions to get a discount: grad year>2017,university=SJSU, degree=graduate.

**CODE:**

'use strict';

var x=20;

function checkIfDiscountApplicable(obj1,obj1,obj2){

if(obj1.university=="SJSU" && obj1.degree=="graduate"){

if(obj2.year>2017){

console.log("discount is applicable");

}

else{

console.log("discount is not applicable");

}

}

else{

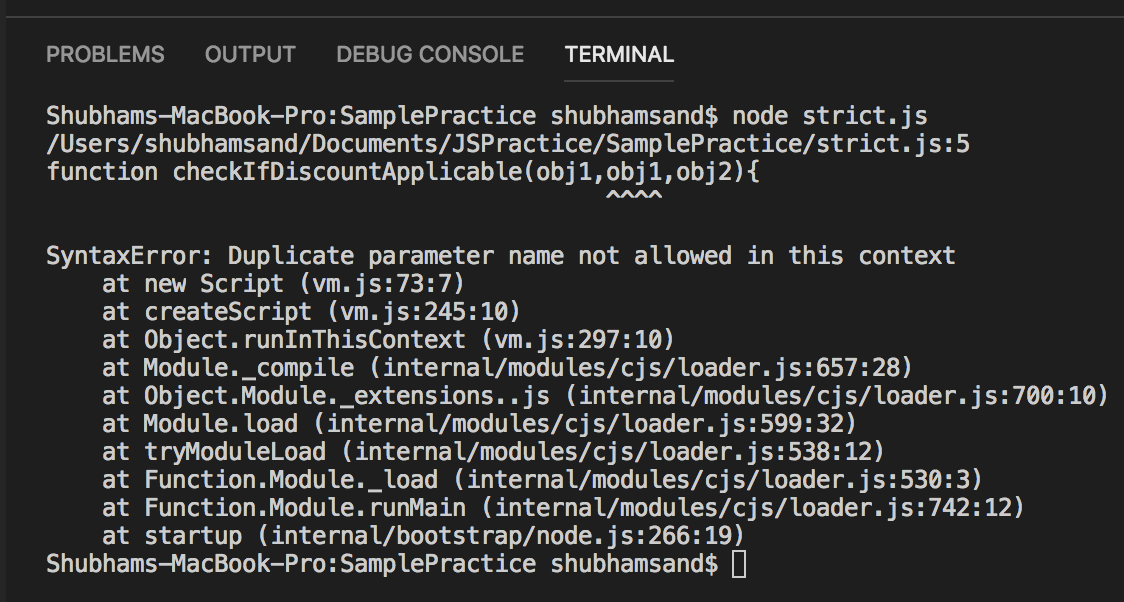
console.log("discount is not applicable");

}

}

ComputeMultiply({university:"SJSU",city:"San Jose"},{degree:graduate},{year:2018});

**OUTPUT**: Error due to duplicate parameter declaration in the function checkIfDiscountApplicable



**6)ERRORS**

* Errors can be reported by JavaScript engine or can be explicitly thrown by the programmer if a particular logical condition is not met.
* There are different types of errors such as SyntaxError, RangeError, TypeError, File Error etc.
* Error() is an object having properties: message ,name of message and it is based from the Error.prototype chain.
* Errors can be handled by using the try{},catch{} and the finally blocks.
* Errors are thrown from try block and handled in the catch block.
* Finally block runs irrespective of error occurred or not.

**PROBLEM STATEMENT**:

String is valid if it has ‘@’ sign present on the left-hand side or the right-hand side of a character. For example, string “ @a@B@U@ “ is valid string whereas string ‘a@b@r@’ is not valid. If string is not valid throw an error and report the error message. Input String will only contain characters and ‘@’ symbols in it.

**CODE:**

function checkIfValid(str) {

var regex1 = /^[a-zA-Z]/,regex2 = /[^@][a-zA-Z]/,regex3 = /[a-zA-Z][^@]/;

try{

if (regex1.test(str)) {

throw new Error("Invalid String");

} else if (regex2.test(str)) {

throw new Error("Invalid String");

} else if (regex3.test(str)) {

throw new Error("Invalid String");

} else {

console.log("The string matched successfully");

}

}catch(e){

console.log("In catch.."+e.message);

}

finally{

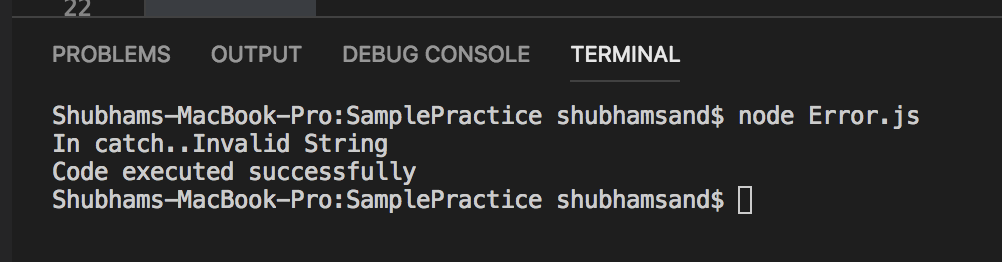
console.log("Code executed successfully");

}

}

checkIfValid("@e@T@/U");

**OUTPUT:**



**7)DEFAULT PARAMS:**

* Default parameters are specified in the function definition and are useful when number of arguments that are passed to a function are less than number of parameters that are expecting a value. If less arguments are passed and no of parameters are more all the extra parameters will get a value called **undefined** which may lead to unexpected results.
* Default values get assigned to parameters during the function call.
* Default values can also be specified during array or **object destructuring.**

**PROBLEM STATEMENT:**

Compute the amount the employee gets in dollars. Employee object holds a special property called incentive in % which impacts the total amount the employee gets. If the property is not specified in the object, all such employees should get an incentive of 10%.

**CODE:**

function computeSalary(employee){

//Specifying default values

var{name,baseSalary=100000,incentive=5}=employee; //Array destructuring

var finalSalary=(baseSalary\*(100+incentive))/100;

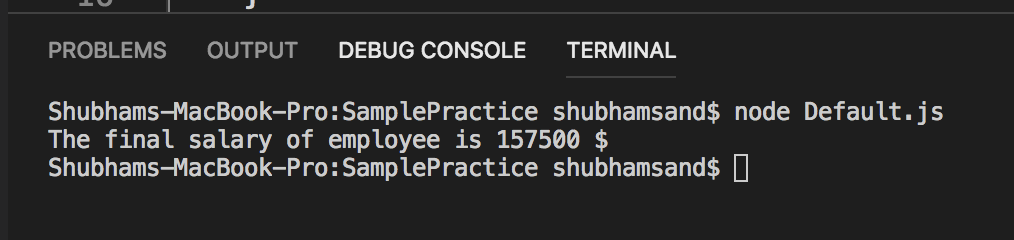
return finalSalary;

}

EmployeeSalary=computeSalary({name:"shubham",baseSalary:150000});

console.log("The final salary of employee is" +" "+EmployeeSalary+" "+"$");

**OUTPUT:**



**8)Includes and typeof:**

* **Includes: includes()** is a method that is used to check if a particular string is present in a given string or if an element is present in an array.
* The function returns **true** if string or an element is present else it returns **false**.
* Syntax: string1.includes(“searchString”);
* **typeof():** This operator is used to determine the type of value the variable is holding.
* Different types in JavaScript include 1)Primitive types which hold a single value like Number, String, undefined etc. 2)Object🡪 for function, RegExp, Object etc.
* Syntax for typeof: typeof(variableName);
* typeOf operator returns function for a function even though it is of type Object.

**PROBLEM STATEMENT:**

Check if a particular city is present in array of cities and if its present return appropriate message.

Input=["San Jose","Modesto","Santa Cruz"],"Modesto"

**OUTPUT:** ”The city is available in list of cities”

**CODE:**

function checkCity(cities,searchCity){

var i;

if(typeof(searchCity)=="string"){ //Checking the type==string

if(cities.includes(searchCity)){ //check if city is present in array

return true;

}else{

return false;

}

}else{

return false;

}

}

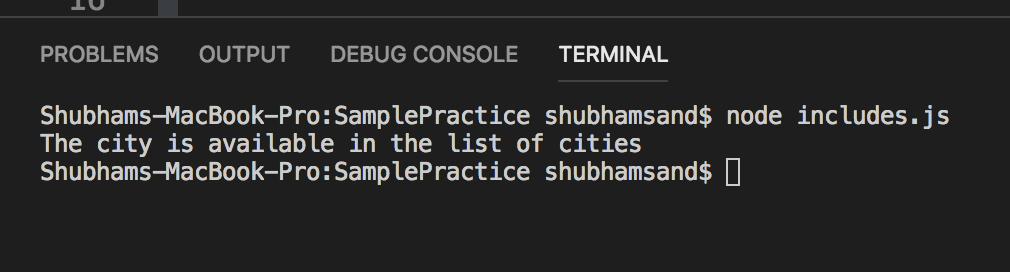
if(checkCity(["San Jose","Modesto","Santa Cruz"],"Modesto")){

console.log("The city is available in the list of cities");

}else{

console.log("City is not present");

}

**OUTPUT:**

**9)Use of import and export:**

* **module.exports** is used to expose JavaScript functions , variables etc. that can be used in another JavaScript code by use of import.
* In node.js **require** is used instead of import.
* In require the path of JavaScript file is mentioned whose functions or variable are to be used.

**PROBLEM STATEMENT:**

Write a JavaScript code to print all the numbers within the 2 specified min and max limits. The function to be implemented is placed is a separated module and used in another file to show demonstration of export and require.

**CODE:**

**EXPORT FILE:**

var eval=function evalExpression(x1,x2){

let res=[];

let i=0;

if(x1>x2){

for(i=x2;i<x1;i++){

res.push(i);

}

res.push(x1);

}else{

for(i=x1;i<x2;i++){

res.push(i);

}

res.push(x2);

}

return res;

}

module.exports=eval;

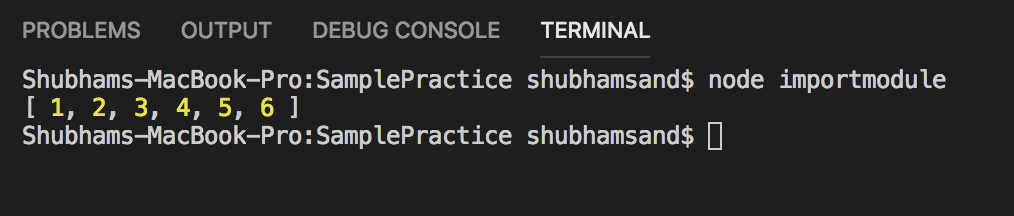
**IMPORT FILE:**

var evalExpression=require('./comptuteValue.js');

var findRange=evalExpression(1,6);

console.log(findRange);

**OUTPUT:**

****

**10)Type Conversion:**

* It is the process of converting a variable from one type to another which can be either done by the programmer can be done implicitly by the JavaScript engine.
* Implicit conversions are done by the engine where it predicts that such type conversion is needed and hence does it while in other cases its solely upon the discretion of the programmer.
* Different methods that are used for explicit type conversion are String(),Number(),Boolean() etc.

**PROBLEM STATEMENT:**

Compute the simple interest for the given principal amount, time and rate of interest. Where time is passed as string.

Formula=(p\*r\*t)/100

Input=Amount=1000,rate=10%,time=”5”

Output=The simple interest to be paid is 500.

**CODE:**

function computeSimpleInterest(p,r,t){

let principal=Number(p);

let rate=Number(r);

let time=Number(t); //Explicit conversion from string to number

let si=(principal\*rate\*time)/100;

return si;

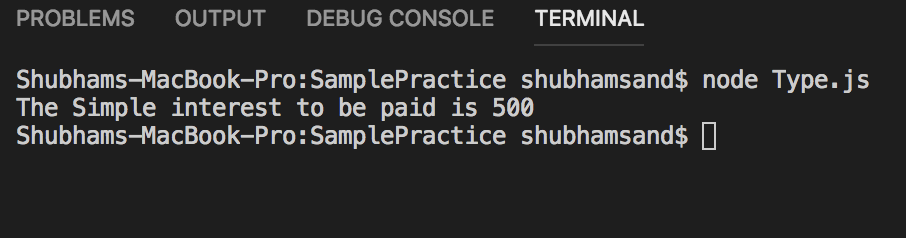
}

var si=computeSimpleInterest(1000,10,"5"); //typeof si=Number

/// Implicit conversion from number to string

console.log("The Simple interest to be paid is"+" "+si);

**OUTPUT:**



**11)JSON:**

* JSON stands for JavaScript Object notation and is generally used to transfer data to and from the client and the server in secured manner.
* Earlier XML was the preferred way of format but JSON being easy to use and also being lightweight became the preferred choice.
* The Object notation and JSON notations are different as in JSON the name of the name:value pair always has to be in double quotes which is not the case in JavaScript Object.
* JSON has 2 important methods 1)parse()🡪used to convert JSON string to JavaScript Object and 2)stringify()🡪to convert to JSON string.

**PROBLEM STATEMENT:**

Update the JSON string having **amountDue** property to 10000 and return back the updated JSON String.

Input: var text = {"name":"Alex","admittedOn":"2018-08-31T23:29:58.163Z","amountDue":1000}

Output={"name":"Alex","admittedOn":"2018-08-31T23:29:58.163Z","amountDue":10000}

**CODE:**

var updateAmount=(str)=>{

//convert JSON string to JavaScript OBject

var obj=JSON.parse(str);

obj.amountDue=10000;

//console.log(obj);

//convert JavascriptObject to JSON string

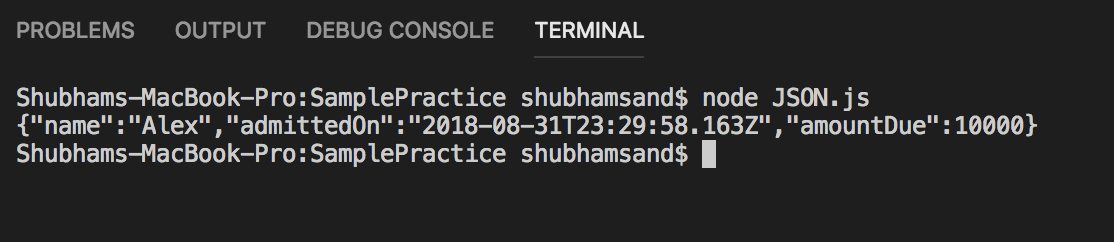
var JSONstr=JSON.stringify(obj);

return JSONstr;

};

console.log(updateAmount('{"name":"Alex","admittedOn":"2018-08-31T23:29:58.163Z","amountDue":1000}'));

**OUTPUT:**



**12)Objects and Classes**

* JavaScript Objects are collection of named value pairs and methods that operate on the data present in the object.
* Objects can be created using the constructor function ,new Object() or using{name:value}.
* Classes bind data and methods together and offers functionalities such as class inheritance.
* Classes are defined using the class keyword and are similar to functions except that we cannot access the class before it is defined.
* A class contains only one method called the constructor which is called when the class is instantiated and sets the properties of the class.

**PROBLEM STATEMENT:**

Calculate the total bill for the the rental car hired .

Formula: (miles\*(Rate/mile))-discount

Input: miles=100 ,rate= 16, discount=200$

Output=1400$

**CODE:**

class miles{

constructor(mile,rate,discount){ //class constructor

this.mile=mile;

this.rate=rate;

this.discount=discount;

}

calculateAmt(){ //class method

return (this.mile\*this.rate)-this.discount;

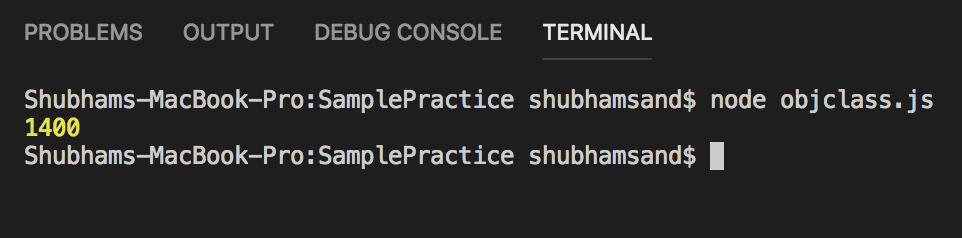
}

}

let m1 = new miles(100,16,200); //class instantiation

console.log(m1.calculateAmt());

**OUTPUT:**



**13)Object.assign**

* Object.assign() is used to calculate values and properties from one or more objects to a destination object.
* Syntax Object.assign(target\_obj, source object 1,source object 2….)
* Object.assign only copies properties that belong to source object and not the properties that are inherited from the prototype chain.

**PROBLEM STATEMENT:**

Collect the entire voter information present in different objects into one object and display the accumulated voter information.

Input:{firstname:”Shubham”,lName:”sand”,city: ”San Jose”}

{voterNum:123065,county:”Santa Clara”}

{state:CA}

**CODE:**

function mergeObjects(obj1,obj2,obj3){

//merge voter information into one object

var voterInformation=Object.assign({},obj1,obj2,obj3);

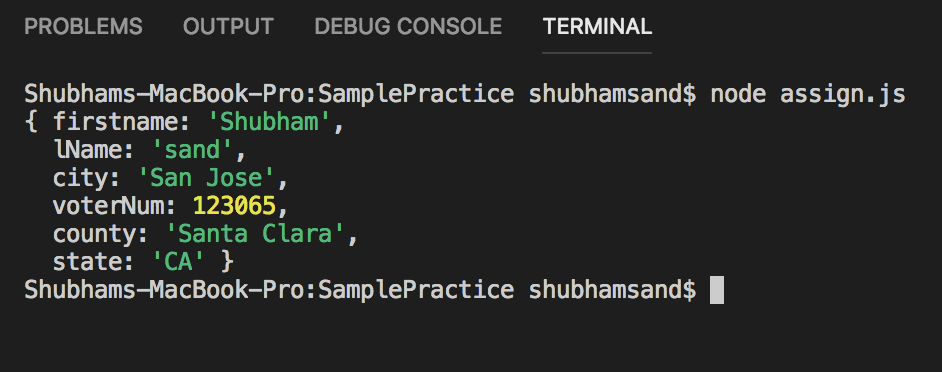
return voterInformation;

}

var obj=mergeObjects({firstname:'Shubham',lName:'sand', city: 'San Jose'},{voterNum:123065,county:'Santa Clara'},{state:"CA"});

console.log(obj);

**OUTPUT:**



**14)STATIC METHOD:**

* Static methods are called by the class name and not by the instance of a class.
* Therefore, if a static method is called by instance of a class it will return undefined.
* Syntax to define static method is static method\_Name{}

**PROBLEM STATEMENT:**

Accept length, breadth, height of 2 cubes and compare volumes of two cubes and return highest volume among the two.

Input v1=10,20,10 v2=5,10,5

Output=2000

**CODE:**

class volume{

constructor(length,breadth,height){

this.length=length;

this.breadth=breadth;

this.height=height;

}

static compareVolume(v1,v2){

var v1=v1.length\*v1.breadth\*v1.height;

var v2=v2.length\*v2.breadth\*v2.height;

if(v1>v2){

return v1;

}

else{

return v2;

}

}

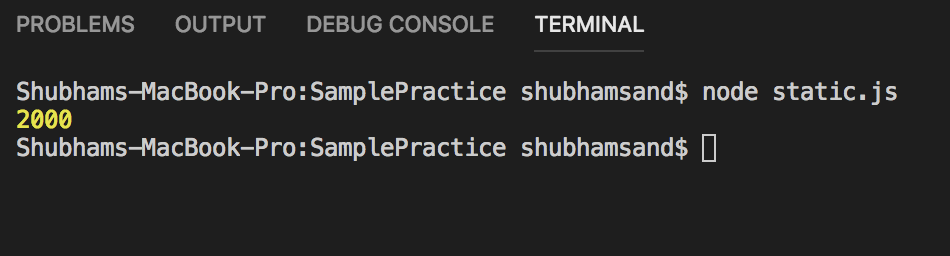
}

const volume1=new volume(10,20,10);

const volume2=new volume(5,10,5);

console.log(volume.compareVolume(volume1,volume2));

**OUTPUT:**



**15)Inheritance using sub-classes in Java Script.**

* In class inheritance the child class or sub class has access to data and methods of its parent class.
* The sub class **extends** parent class to depict inheritance among them.
* The constructor in sub class first calls the constructor in super class using the keyword **super()** and then continues execution.

**PROBLEM STATEMENT:**

Implement a functionality of a LMV car which inherits basic functionalities like drive(),brake/() etc. from the parent Car class.

**CODE:**

class Car{

constructor(name,make){

this.name=name;

this.make=make;

}

break(){

console.log("Car is braking");

}

drive(){

console.log("Car is moving");

}

}

class LMV extends Car{ //LMV extends car-->inheritance

constructor(name,make){

super(name,make);

}

sportsMode(){

console.log("Car is in sports mode");

}

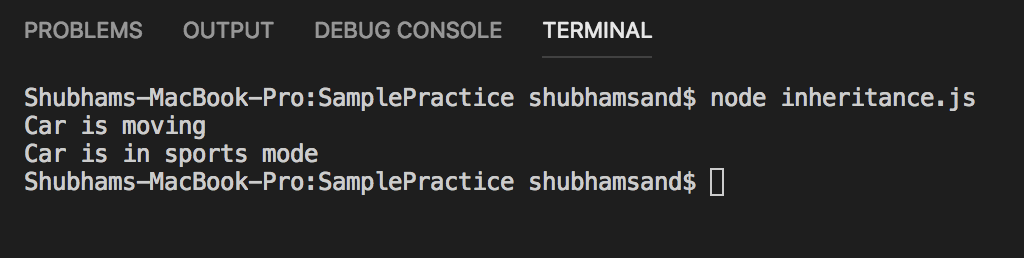
}

let c=new LMV("Accord","Honda");

c.drive();

c.sportsMode();

**OUTPUT:**



**16)Method Overriding:**

* When multiple functions are defined in JavaScript having same names and when the function call is made to that function name then the last one defined will override all the previous functions and always the last defined function will be executed.

**PROBLEM STATEMENT:**

Accept the values and calculate the area of the figure. Return the area thus calculated. Define methods to calculate the area if rectangle and square.

**CODE:**

function Area(side1,side2){ //constructor function

this.side1=side1;

this.side2=side2;

}

Area.prototype.calArea=function(){ //area of rectangle

return this.side1\*this.side2;

}

Area.prototype.calArea=function(){ //overided method area of asquare

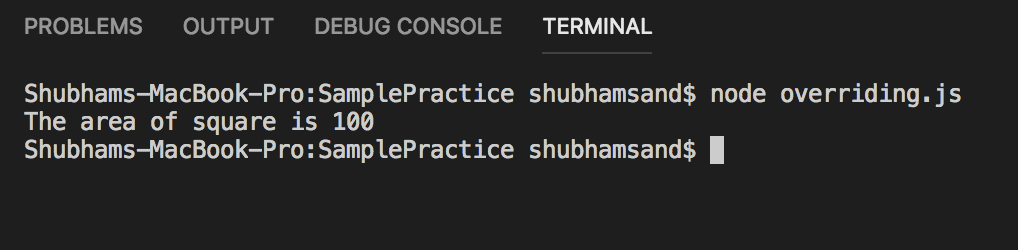
return this.side1\*this.side1;

}

var a1=new Area(10,20);

console.log("The area of square is"+ " "+a1.calArea());

**OUTPUT:**



**17)Use of get(In classes):**

* get or Getter method is used to get some value using the object properties. More so the object property itself.
* Therefore, when a property is looked for ,then the corresponding getter is called upon to return and access that value.

**PROBLEM STATEMENT:**

Calculate the surface area of the square and return the surface area using the get method. The class constructor accepts side as parameter.

Input: side=4

Output=16

**CODE:**

class surfaceArea{

constructor(side){

this.side=side;

}

get sArea(){ // getter method to get surface area

return 4\*this.side;

}

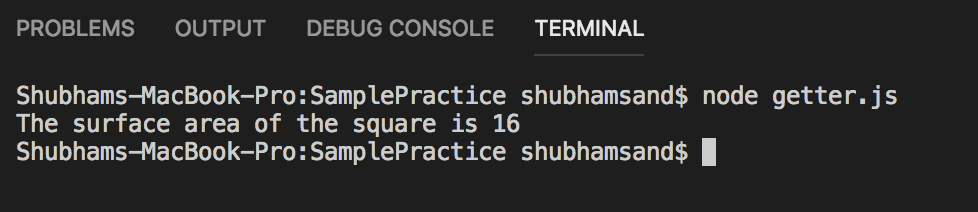
}

var s1=new surfaceArea(4);

//call to getter

console.log("The surface area of the square is"+" "+ s1.sArea);

**OUTPUT:**



**18)fetch()**

* fetch() method is used to get data from an URL.
* It returns a response object and uses Promises to avoid hefty callbacks.
* The response object then can be used to get data when promise resolves or handle exception when promise is rejected.
* Syntax fetch(URL, options).

**PROBLEM STATEMENT:**

Using an API for colors display the hexadecimal values of various colors.

API: <http://www.colr.org/json/colors/random/7>

**CODE:**

var a;

fetch('http://www.colr.org/json/colors/random/7')

.then(response => response.json())

.then(json => {

console.log(json.colors);

// a=JSON.parse(json);

document.getElementById("demo").innerHTML=json.colors.map(col => col.hex);

});

**HTML:**

<html>

<head>

<title>Colors</title>

</head>

<body>

<h3>Display Color details</h3>

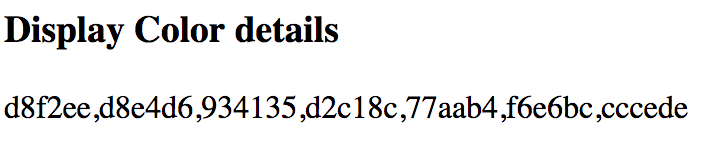
<p id="demo"></p>

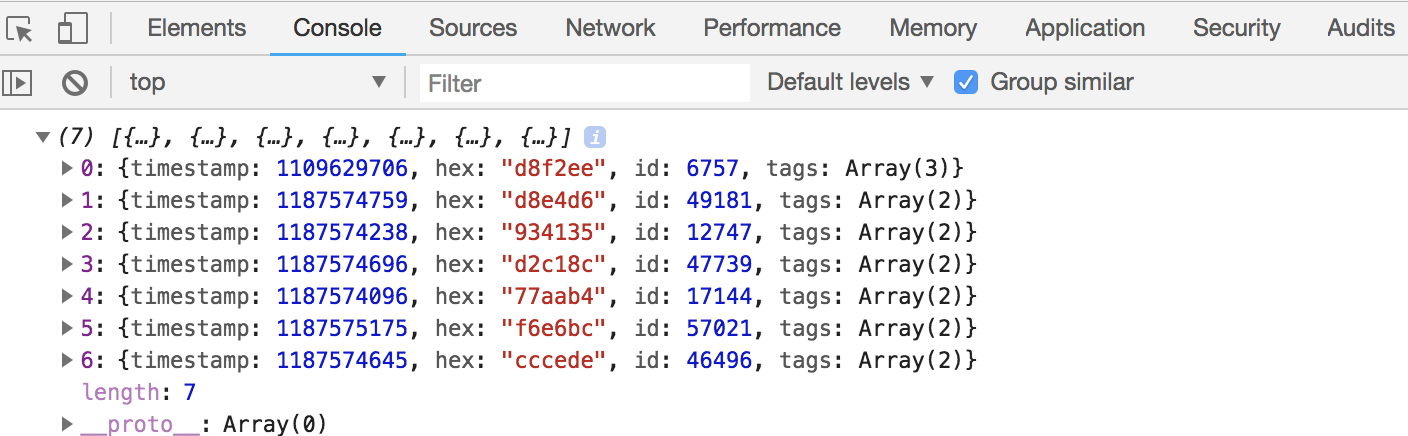
<script src="fetch.js"></script>

</body>

</html>

**OUTPUT:**





**HTML5**

**1)Local Storage:**

* HTML local storage helps to store data in the browser which lasts even if the browser is closed.
* Earlier cookies were used which provided less storage up to 4KB unlike local storage which can store data locally from 5MB to 10MB.
* The local storage data can be accessed by pages of same domain and protocol that created the localStorage object.
* All local storage properties and data are saved in form of key-value pairs.

**PROBLEM STATEMENT:**

Display the name of authors already stored in an array. Then add few more author names to array and store the array as Local storage and close the browser window. Then again open the browser window to print all the added author names to check if already added author names persists or not.

**CODE:**

<html>

<head><title>Display authors</title></head>

<body>

<script>

var arr=["J.K Rowling","Mark Twain","Dr.Seuss","Jane Austen","XYZ"];

var aut=[];

function showAuthors(){

//display from local storage

document.getElementById("authors").innerHTML=localStorage.authors;

}

function addAuthors(){

aut.push(arr.pop());

//add to local storage

localStorage.authors=aut.toString();

}

</script>

<div>

<button onclick="showAuthors()">Click to Display authors</button>

<button onclick="addAuthors()">Add Authors</button>

<h3>List of authors is:</h3>

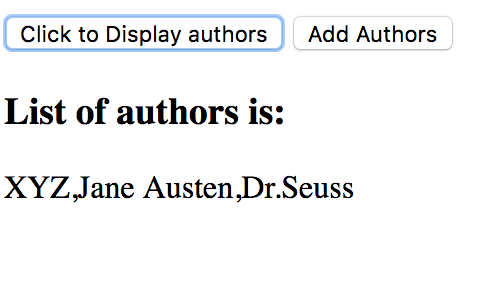
<p id="authors"></p>

</div>

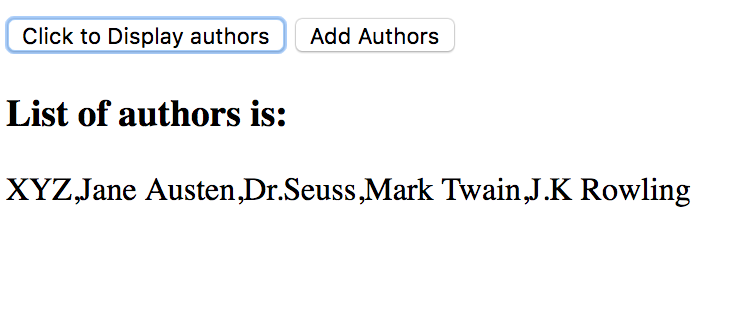
</body>

</html>

**OUTPUT:**



Few authors were added, and browser was closes and opened again to display all added authors:



**2)Media(Audio and Video)**

* Audio: **<audio>** tag is used to specify and play an audio on an HTML web page.
* **controls** attribute is used to add features like playing ,volume control etc.
* **<source>** element in used inside the <audio> tag to specify the location of audio files. We can specify more than one files but only the 1st compatible file will be added.
* Audio files like mpeg , ogg and wav are supported.
* Video**:<video>** tag is used to specify the video with same **controls** attribute like the audio and the **<source>** element to specify the source video files.
* Supported video files are webM , MP4, ogg.

**PROBLEM STATEMENT:**

Add a video file and set width =300 px and height=400 px of the video screen. Add an audio file using audio tag and its corresponding attribute and elements.

**CODE:**

<html>

<head><title>Audio and Video Comtrols</title></head>

<body>

<div>

<h3>Video</h3>

<video width="500" height="400" controls>

<source src="SampleVideo\_1280x720\_1mb.mp4" type="video/mp4">

</video>

</div>

<div>

<h3>Audio</h3>

<audio controls>

<source src="SampleAudio\_0.4mb.mp3" type="audio/mpeg">

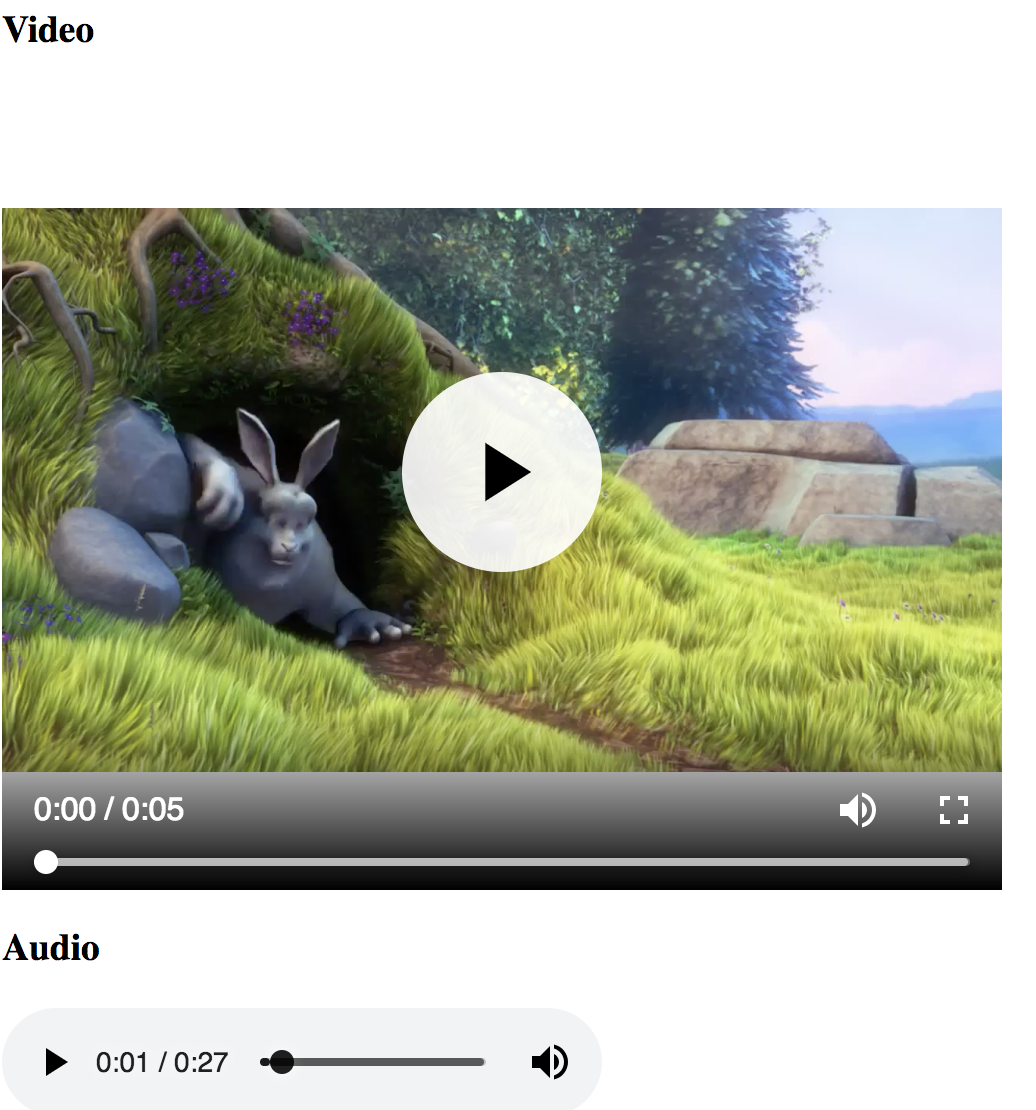
</audio>

</div>

</body>

</html>

**OUTPUT:**



**3)Input types:**

* Input types are used to accept input data from the user usually inside interactive forms.
* Various Input Types are as follows:

1. Input type password: User inputs password in this field and the typed text is encrypted.

* Pattern attribute: to match the password for a particular regular expression.
* Placeholder: specifies the hint that can be used to enter the password.

1. Input type text: Accepts user input as a string.

* value attribute: to specify default value to the input text field.
* disabled: specifies the input field cannot be used and cannot be clicked.
* maxlength: specifies the maximum number of characters that can be entered.
* autofocus: places the cursor to this field to default on the page.
* width and height: to specify the dimensions of the input field.
* Required: specifies that value for the field should be entered.

3)Input type submit: To submit the form data to the server via handler.

4)Input type radio: Used to specify the radio buttons

* checked: to check the radio button by default.
* Name: to display name of the radio button.
* Value: the value that would be taken when that radio button is checked.

5)Input type email : To match the text to a valid email address.

6)Input type number: To accept only numeric input.

* Min: specifies the minimum value that should be entered.
* Max: maximum value that should be entered.

7)input type reset: to reset the form values.

**PROBLEM STATEMENT:**

Design a registration form to open an account for rental car services. Use the HTML input types to accept and validate the data.

**CODE:**

<html>

<head><title>XYZ Car Account opening</title></head>

<body>

<h3>XYZ Car Account opening</h3>

<form action="" autocomplete="on">

First name:<input type="text" name="FirstName" autofocus required><br><br>

Last name: <input type="text" name="LastName" placeholder="Enter Last Name" required><br><br>

User Name: <input type="text" name="Use Name" maxlength="10" pattern="[a-zA-z0-9]" required><br><br>

Password: <input type="password" name="password" placeholder="Enter strong password" required><br><br>

Frequency of Use: <input type="number" name="frequency" min="1" width="10" height="5"><br><br>

E-mail: <input type="email" name="email"><br><br>

Get offers via email: <input type="radio" name="allow" value="allow" checked> <input type="radio" name="No" value="No"><br><br>

<input type="submit" value="Submit">

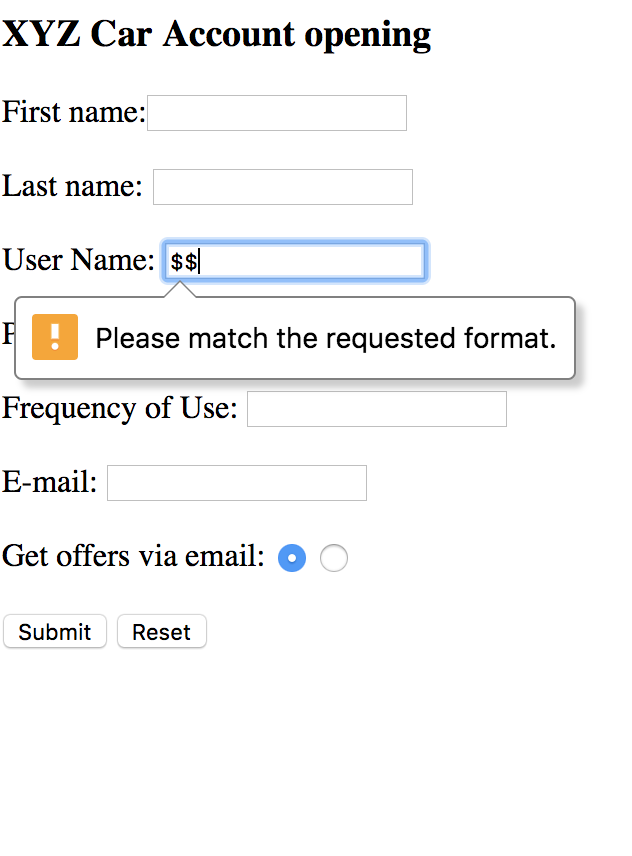
<input type="reset">

</form>

</body>

</html>

**OUTPUT:**

  
**4)HTML Geolocation:**

* Geolocation is used to get the actual location of the user.
* getCurrentPosition() method is used to return the current position object of the user.
* This returned object has default properties like cords.latitude ,coords.longitude etc. which can be used to determine the location.
* watchPosition() method always returns the current position object of the user as he moves.
* Whenever geolocation asked for the user has the right to accept the location sharing or decline the request thus protecting user privacy.

**PROBLEM STATEMENT:**

Show the current location of the user in coordinates of longitude ,latitude and altitude.

**CODE:**

<html>

<head></head>

<body><h2>Display Users current location</h2>

<input type="text" onmouseover="getLocation()" name="Get Current Location" value="Show Location" style="background-color:cornflowerblue"><br><br>

<p id="location"></p>

<script>

function getLocation(){

function recieved(location){

let lat=location.coords.latitude;

let longt=location.coords.longitude;

let alt=location.coords.altitude;

document.getElementById("location").innerHTML="Latitude is"+" "+lat+" "+"Longitude is"+" "+longt+" "+"altitude is"+" "+alt;

}

function reject(location){

document.getElementById("location").innerHTML="Location not found";

}

navigator.geolocation.getCurrentPosition(recieved,reject);

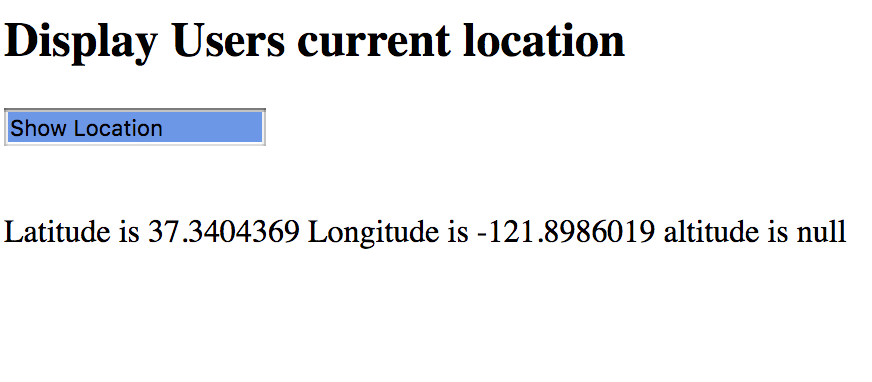
}

</script>

</body>

</html>

**OUTPUT:**



**JAVA**

**1)Queue**

* Java Queue is an interface based on the collections framework and uses the first in first out principle to insert or remove elements.
* Operations supported by queue are enqueue(adding elements) dequeue(removing elements).
* Elements are deleted from head end and elements are added at rear end.

**PROBLEM STATEMENT:**

Implement a Java queue to support operations enqueue, dequeue and test the operations using Junit framework.

**CODE:**

**QueueOperations.java**

**package** QueuePackage;

**import** java.util.Queue;

**import** java.util.Scanner;

**public** **class** QueueOperations {

**public** **int** enqueue(Queue<Integer> q1) {

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter element to be added");

**int** a=sc.nextInt();

q1.add(a);

**return** a;

}

**public** **int** dequeue(Queue<Integer> q1) {

**int** a=q1.remove();

System.***out***.println("Element removed is"+a);

**return** a;

}

**public** **int** sizeOfQueue(Queue<Integer> q1) {

**int** size=q1.size();

**return** size;

}

}

**QueueOps.java**

**package** QueuePackage;

**import** java.util.LinkedList;

**import** java.util.Queue;

**public** **class** QueueOps {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

QueueOperations q= **new** QueueOperations();

Queue<Integer> q1 = **new** LinkedList<>();

**int** a=q.enqueue(q1);

System.***out***.println("The element enqueued in queue is "+a);

**int** b=q.enqueue(q1);

**int** c=q.dequeue(q1);

System.***out***.println("The element dequeued from the queue is "+c);

System.***out***.println("The size of the queue is "+ q.sizeOfQueue(q1));

}

}

**TestQueue.java---Junit**

**package** QueuePackage;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**import** **static** org.junit.jupiter.api.Assumptions.*assumingThat*;

**import** java.util.LinkedList;

**import** java.util.Queue;

**import** java.util.Stack;

**import** org.junit.jupiter.api.Test;

**class** TestQueue {

@Test

**public** **void** testEnqueue(){

QueueOperations q= **new** QueueOperations();

Queue<Integer> q1 = **new** LinkedList<>();

**int** a=q.enqueue(q1);

*assertEquals*(10,a);

}

@Test

**public** **void** testDequeue(){

QueueOperations q= **new** QueueOperations();

Queue<Integer> q1 = **new** LinkedList<>();

**int** b=q.enqueue(q1);

**int** a=q.dequeue(q1);

*assertEquals*(10,a);

}

@Test

**public** **void** testsize(){

QueueOperations q= **new** QueueOperations();

Queue<Integer> q1 = **new** LinkedList<>();

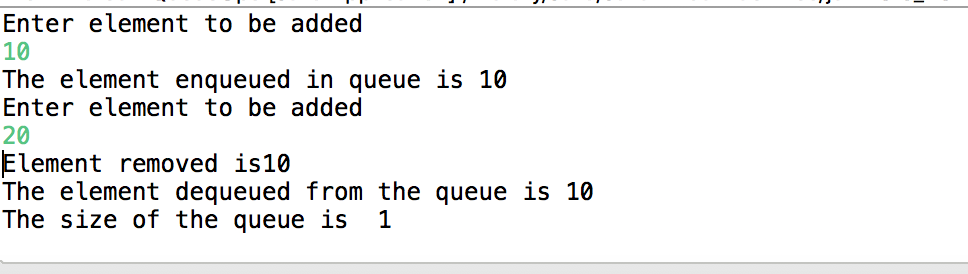
**int** a=q.sizeOfQueue(q1);

*assertEquals*(0,a);

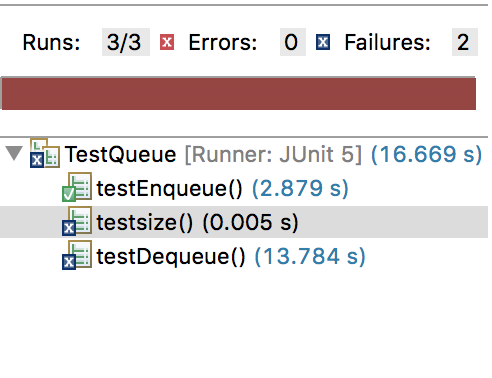
}

}

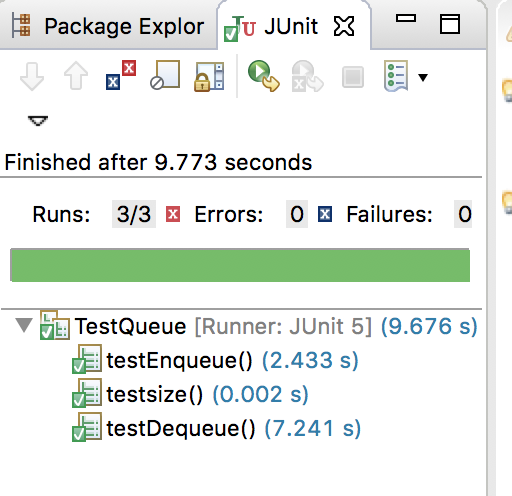
**OUTPUT:**

****

Failed test cases for wrong result checking: 1 passed 2 failed



All passed test cases:



**2)Stack:**

* Java stack inherits from vector and is part of the collections framework of Java.
* Stacks are based on last in first out principle.
* Stacks in java support basic stack operation like push, pop, peek, search.

**PROBLEM STATEMENT:**

Implement the stack in java to support basic operations of push,pop,search and build Junit test cases to automate testing.

**CODE:**

**StackOps.java**

**package** Stack;

**import** java.util.Scanner;

**import** java.util.Stack;

**public** **class** StackOps {

**public** **int** push(Stack<Integer> s1) {

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter element to be pushed");

**int** a=sc.nextInt();

s1.push(a);

**return** a;

}

**public** **int** pop(Stack<Integer> s1) {

Integer p=(Integer) s1.pop();

System.***out***.println("The poped out element is"+p);

**return** p;

}

**public** **int** search(Stack<Integer> s1) {

Scanner sc=**new** Scanner(System.***in***);

System.***out***.println("Enter the element to search :");

**int** search=sc.nextInt();

Integer loc=(Integer) s1.search(search);

**if**(loc!=-1) {

System.***out***.println("The element is found at location" +loc);

**return** loc;

}

**return** loc;

}

}

**Execute.java**

**package** Stack;

**import** java.util.Stack;

**public** **class** Execute {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Stack<Integer> s1 = **new** Stack<Integer>();

StackOps so=**new** StackOps();

**int** a=so.push(s1);

System.***out***.println("Element pushed to stack is "+ a);

**int** b=so.push(s1);

System.***out***.println("Element pushed to stack is "+ b);

**int** c=so.pop(s1);

System.***out***.println("Element popped from stack is "+ c);

**int** d=so.search(s1);

System.***out***.println("Location of element seached is "+d);

}

}

**Junit test file : checkStackOperations.java**

**package** Stack;

**import** **static** org.junit.Assert.\*;

**import** java.util.Stack;

**import** org.junit.Test;

**public** **class** CheckStackOperations {

@Test

**public** **void** testPop() {

StackOps test=**new** StackOps();

Stack<Integer> s = **new** Stack<Integer>();

**int** a=test.push(s);

**int** num= test.pop(s);

*assertEquals*(10,num);

}

@Test

**public** **void** testSearch() {

StackOps test=**new** StackOps();

Stack<Integer> s = **new** Stack<Integer>();

**int** a=test.push(s);

**int** ele=test.search(s);

*assertEquals*(1,ele);

}

@Test

**public** **void** testPush() {

StackOps test=**new** StackOps();

Stack<Integer> s = **new** Stack<Integer>();

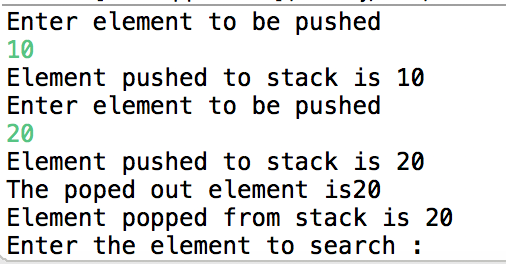
**int** a=test.push(s);

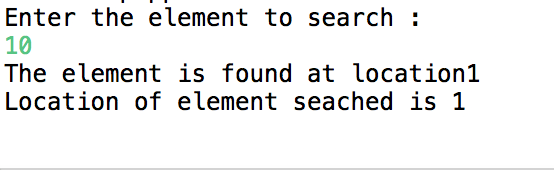
*assertEquals*(10,a);

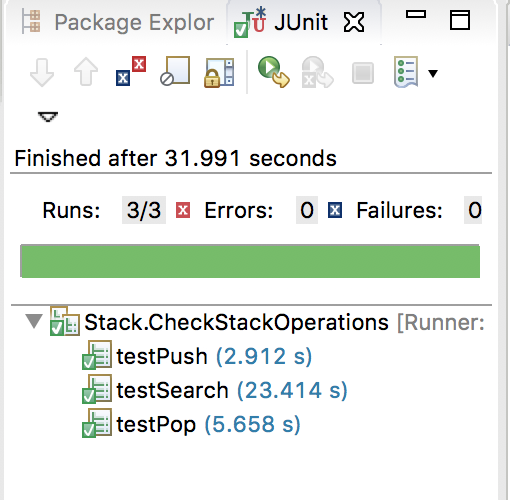
}

}

**OUTPUT:**

****

****

****

**3)Arrays:**

* Array in Java is used to store homogeneous elements like array of integers or array of strings etc.
* Array indexes elements starting from 0 up to array\_name.length-1.
* Array declaration and memory space allocation is done by the syntax:

int [] arr\_name=new int[10];

* Arrays can be single dimensional or multidimensional. Multidimensional meaning values are stored in form of rows and columns where each element holds address of other array.

**PROBLEM STATEMENT:**

Write a Java program to calculate the maximum frequency of an element in the array. Write Junit Test case to test the results.

**CODE:**

**ArrayOperations.java**

**package** arrayPack;

**public** **class** ArrayOperations {

**public** **int** MaxFrequency(**int**[] arr) {

**int** Hfrequency=0;

**for** (**int** i = 0; i < arr.length; i++) {

**int** count = 0;

**for** (**int** j = 0; j < arr.length; j++) {

**if** (arr[i]==arr[j])

count++;

}

**if** (count >Hfrequency)

Hfrequency = count;

}

**return** Hfrequency;

}

}

**MainArray.java**

**package** arrayPack;

**public** **class** MainArray {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int** [] arr=**new** **int**[]{1,1,2,2,2,2,2,3,3,5,6};

ArrayOperations ap= **new** ArrayOperations();

**int** max=ap.MaxFrequency(arr);

System.***out***.println("The maximum frequency in the array is "+max);

}

}

**TestArray.java Junit test cases**

package arrayPack;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class TestArray {

@Test

void testMaxFrequency() {

int [] arr=new int[]{1,1,2,2,2,2,2,3,3,5,6};

ArrayOperations ap= new ArrayOperations();

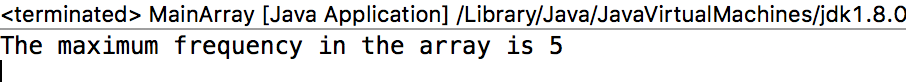
int max=ap.MaxFrequency(arr);

assertEquals(5,max);

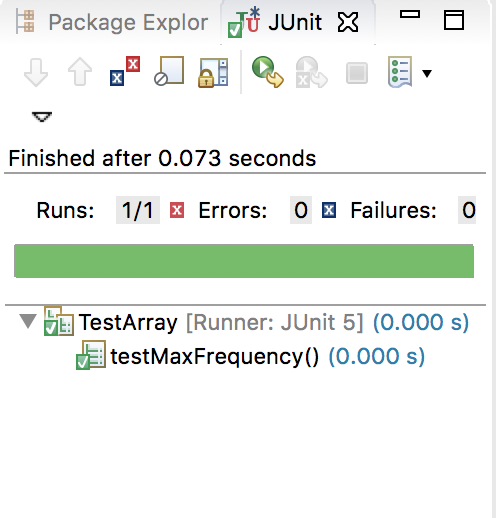
}

}

**OUTPUT:**







**4)Interfaces:**

* Interfaces define abstract methods and final fields. The class which implements the interface must provide function body to all the methods of the interface or else the class will also be an abstract class.
* A class implements and interface with syntax class class\_name implements interface\_name.
* By default, Java doesn’t support multiple inheritance but with use of interface where class can implement more than one interface, it is possible to achieve multiple inheritance.

**PROBLEM STATEMENT:**

Implement a class to calculate the area and perimeter of rectangle. The class should implement an interface where the interface defines the abstract methods for calculating area and perimeter.

**CODE:**

**Figure.java**

**package** InterFace;

**public** **interface** Figure {

**int** calArea(**int** l,**int** b);

**int** calPerimeter(**int** l,**int** b);

}

**Rectangle.java**

**package** InterFace;

**public** **class** Rectangle **implements** Figure {

@Override

**public** **int** calArea(**int** l, **int** b) {

// **TODO** Auto-generated method stub

**int** area=l\*b;

System.***out***.println("Area of Rectangle is"+" "+ area);

**return** area;

}

@Override

**public** **int** calPerimeter(**int** l, **int** b) {

// **TODO** Auto-generated method stub

**int** perimeter=(2\*l)+(2\*b);

System.***out***.println("Perimeter of Rectangle is"+" "+ perimeter);

**return** perimeter;

}

}

**InterfaceMain.java**

**package** InterFace;

**public** **class** InterfaceMain {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

Rectangle r=**new** Rectangle();

r.calArea(10, 20);

r.calPerimeter(10, 20);

}

}

**TestInterface.java**

package InterFace;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class TestInterface {

@Test

void testArea() {

Rectangle r=new Rectangle();

int area=r.calArea(10, 20);

assertEquals(200, area);

}

@Test

void testPerimeter() {

Rectangle r=new Rectangle();

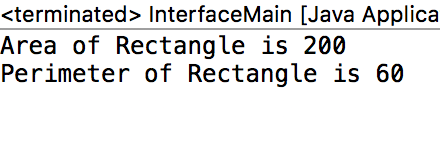
int pm=r.calPerimeter(10, 20);

assertEquals(60, pm);

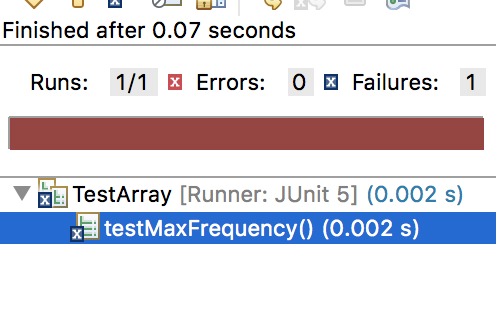
}

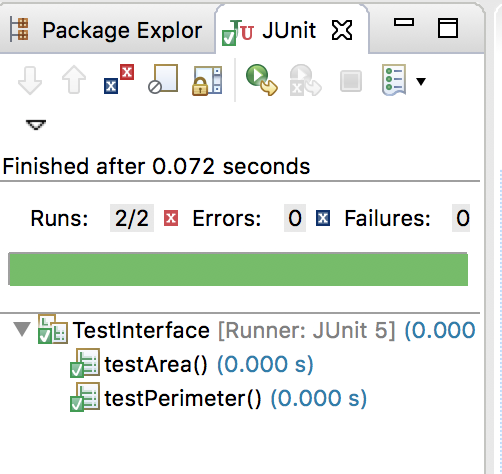
}

**OUTPUT:**

****

**Failed test case for value=3**

****

****

**5)Collections:**

* Collection in Java is an interface and is implemented by various classes such as ArrayList, HashSet, LinkedList to store group of objects.
* This Collection interface provides some of the methods which can be used in all Collections such as add() , remove() ,size() , isEmpty() etc.
* Collection framework also provides an Iterator to traverse the elements of the collection in one direction.
* HashMap’s for instance implement the Map interface which in turn implement the collection interface.
* HashMap’s store the data in form of key: value pairs and there is no ordering of the elements.

**PROBLEM STATEMENT:**

Implement a HashMap to add patient information in a hospital. The key would the patient id and patient information would be name, age, disease. When the patient is discharged he/she should be removed from the HashMap.

**CODE:**

**Patient.java**

**package** map;

**import** java.util.Map;

**public** **class** Patient {

**int** patientId,age;

String name,disease;

**public** Patient(**int** pId,String name,**int** age,String disease) {

**this**.patientId=pId;

**this**.age=age;

**this**.name=name;

**this**.disease=disease;

}

**public** **int** admitPatients(Map<Integer,Patient> map, Patient p) {

map.put(**this**.patientId,p);

**return** **this**.patientId;

}

**public** **int** dischargePatients(Map<Integer,Patient> map,**int** pid) {

map.remove(pid);

**return** pid;

}

}

**HashMapMain.java**

**package** map;

**import** java.util.HashMap;

**public** **class** HashMapMain {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

HashMap<Integer,Patient > map = **new** HashMap<Integer, Patient>();

Patient p=**new** Patient(101, "shubham", 23, "cough");

System.***out***.println("Patient admitted to hospital is "+ p.admitPatients(map,p));

System.***out***.println("Patient with patient id "+p.dischargePatients(map, 101)+" "+"successfully discharged");

}

}

**TestHashMap.java**

**package** map;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**import** java.util.HashMap;

**import** org.junit.jupiter.api.Test;

**class** TestHashMap {

HashMap<Integer,Patient > map = **new** HashMap<Integer, Patient>();

@Test

**void** testAddPatients() {

Patient p=**new** Patient(101, "shubham", 23, "cough");

**int** pid=p.admitPatients(map,p);

System.***out***.println(pid);

*assertEquals*(101, pid);

}

@Test

**void** dischargePatients() {

Patient p=**new** Patient(101, "shubham", 23, "cough");

//HashMap<Integer,Patient > map = new HashMap<Integer, Patient>();

**int** pid=p.dischargePatients(map, 101);

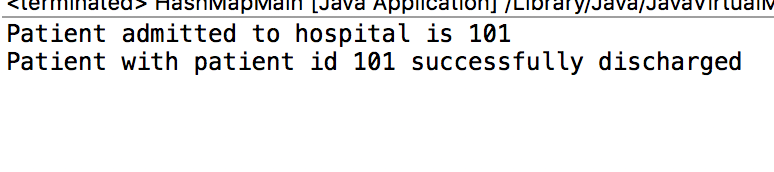
System.***out***.println(map);

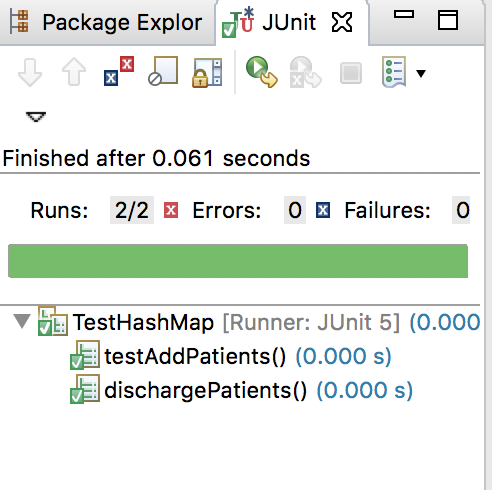
*assertEquals*(101, pid);

}

}

**OUTPUT**





**6)Generics:**

* Generics allow classes and methods to handle roles based on types passed as arguments to them.
* In Generic methods only, parameter of types which are non-primitive can be accepted such as arrays etc. can be passed and accepted.
* Methods return type is preceded by a <T> section.
* In classes the class name is followed by a type<T> section. The <T> specifies that the class would be instantiated with T type like Integer, String etc.
* All this dynamic parameter passing reduces the need of type casting.

**PROBLEM STATEMENT:**

Write a Java code to create a pair with any combination like integer-integer ,integer-string, string-string dynamically based on types of data passed to class constructor. Use Generics to specify the type parameter passing.

**CODE:**

**MyPair.java**

**package** genricPackage;

**public** **class** MyPair<E, T> {

**private** E first;

**private** T second;

**public** MyPair(E e,T t) {

first = e;

second = t;

}

**public** **void** setFirst(E first) {

**this**.first = first;

}

**public** **void** setSecond(T second) {

**this**.second = second;

}

**public** E getFirst() {

**return** first;

}

**public** T getSecond() {

**return** second;

}

}

**GenericMain.java**

**package** genricPackage;

**public** **class** GenericMain {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

MyPair<Integer, String> myPair = **new** MyPair<>(0, "zero");

System.***out***.println("First part of pair is "+myPair.getFirst());

System.***out***.println("Second part of pair is "+myPair.getSecond());;

}

}

**TestGeneric.java**

package genricPackage;

import static org.junit.jupiter.api.Assertions.\*;

import org.junit.jupiter.api.Test;

class TestGeneric {

@Test

void testPair() {

MyPair<Integer, String> myPair = new MyPair<>(0, "zero");

int first=myPair.getFirst();

String second=myPair.getSecond();

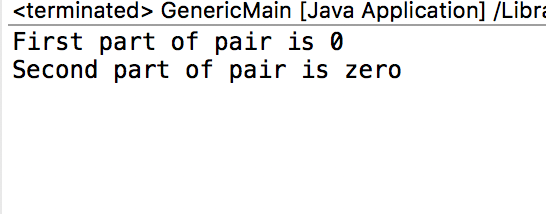
assertEquals(0, first);

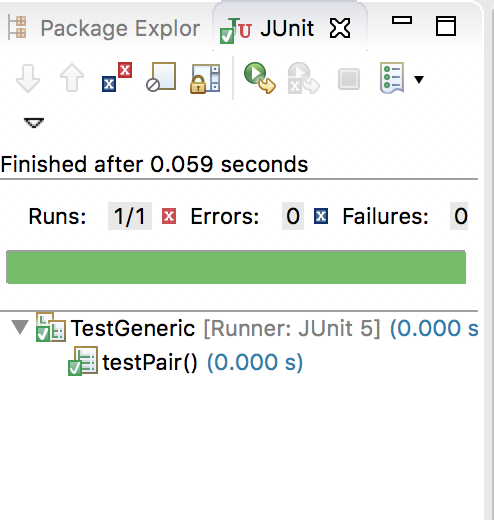
assertEquals("zero", second);

}

}

**OUTPUT:**

****



**7)Multithreading:**

* Multithreading is Java is achieved by creating threads which can operate in parallel to achieve maximum utilization of resources such as memory, CPU processing power etc.
* These threads execute independently and hence do not hinder other threads.
* Multithreading in Java can be achieved by using Thread class or by using Runnable.
* When application is started 1 user thread is created by default.

**PROBLEM STATEMENT:**

Implement Java Code to multiply two numbers and run that operation on a separate thread and then return the result.

**CODE:**

**ThreadOpns.java**

**package** thread;

**import** java.util.concurrent.Callable;

**public** **class** ThreadOpns **implements** Callable<Integer>{

**int** a;

**int** b;

**public** ThreadOpns(**int** a,**int** b) {

// **TODO** Auto-generated constructor stub

**this**.a=a;

**this**.b=b;

}

@Override

**public** Integer call() **throws** Exception {

// **TODO** Auto-generated method stub

**return** **this**.a \***this**.b;

}

}

**MultithreadingMain.java**

**package** thread;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public** **class** MultithreadingMain {

**public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {

// **TODO** Auto-generated method stub

ExecutorService es = Executors.*newSingleThreadExecutor*();

ThreadOpns multiply= **new** ThreadOpns(10, 20);

Future<Integer> future = (Future<Integer>) es.submit(multiply);

System.***out***.println("Multiplication running on thread computes to "+future.get());

}

}

**TestCallable.java**

**package** thread;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**import** org.junit.jupiter.api.Test;

**class** TestCallable {

ExecutorService es = Executors.*newSingleThreadExecutor*();

ThreadOpns multiply= **new** ThreadOpns(10, 20);

Future<Integer> future = (Future<Integer>) es.submit(multiply);

@Test

**void** test() **throws** InterruptedException, ExecutionException {

**int** res=future.get();

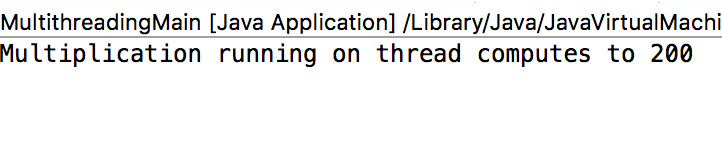
System.***out***.println(res);

*assertEquals*(200, res);

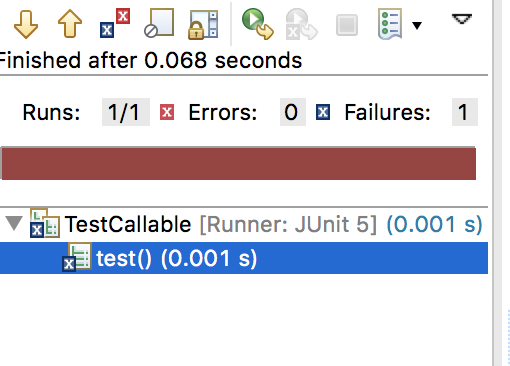
}

}

**OUTPUT:**

****

**Failed test case for res=100**

****

**Passed test case**

