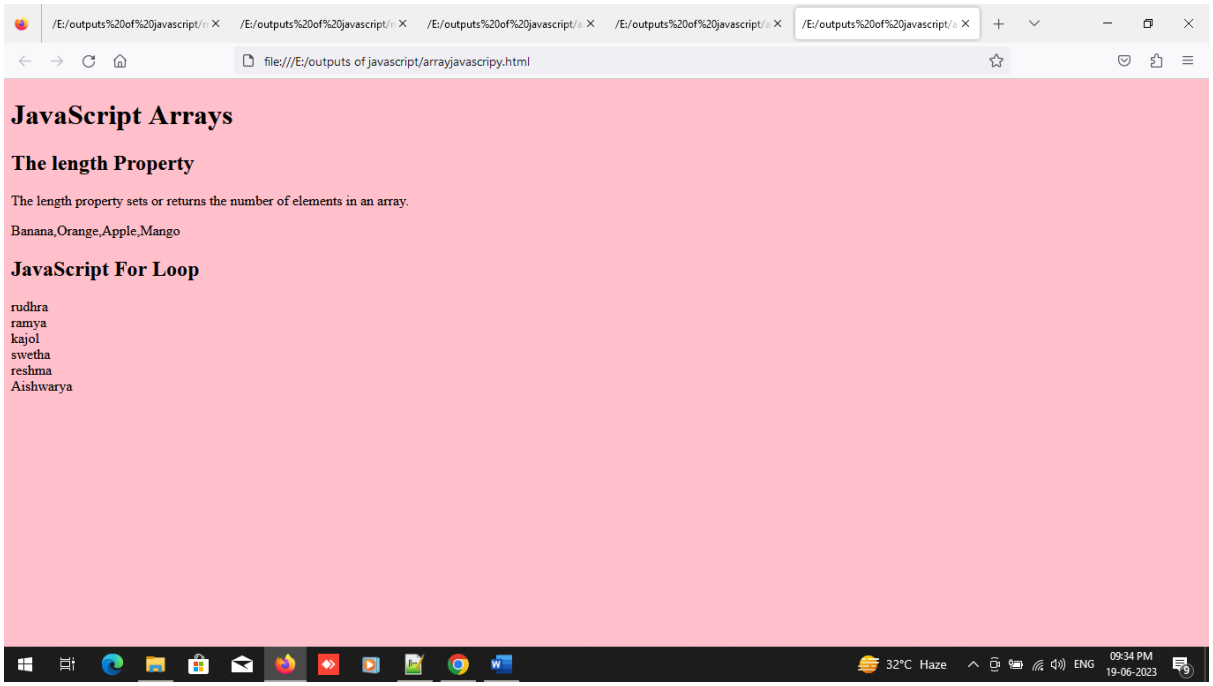
```
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Arrays</h1>
<h2>The length Property</h2>
<body bgcolor="pink";>
<p>The length property sets or returns the number of elements in an array.</p>

<p id="fruits"></p>

<script>
const fruits = ["Banana", "Orange", "Apple", "Mango"];



document.getElementById("fruits").innerHTML = fruits;
</script>
</body>
</html>
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript For Loop</h2>

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

<script>
const names = ["rudhra", "ramya", "kajol", "swetha", "reshma", "Aishwarya"];

let text = "";
```

```
for (let i = 0; i < names.length; i++) {
  text += names[i] + "<br>";
}

document.getElementById("demo").innerHTML = text;
</script>

</body>
</html>
```

Output:




```
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Strings</h1>
<h2>The replace() Method</h2>
```

```
<p>replace() searches a string for a value,
and returns a new string with the specified value(s) replaced:</p>

hello world!



<script>
let text = "hello world!"
 text= text.replace("world", "kajol");
 document.write(text);
</script>

</body>
</html>
<!DOCTYPE html>
<html>
<h1>JavaScript Strings</h1>
<h2>padend</h2>
<body bgcolor="pink">
<script>
var pencil= "8";
document.write(pencil.padEnd(4,"2"));
</script>
<hr>
<h1>javascript charat</h1>
<script>
var rudhra = "she is very beautiful";
document.write(rudhra.charAt("0"));
</script>
<hr>
<script>
```

```
let manju = "she is my mother in law";
document.write(manju.charCodeAt("18"));
</script>
<hr>
<script>
var venkatesh = "he,is,my,father,in,law";
document.write(venkatesh.split("|"));
</script>
</body>
</html>
<!DOCTYPE html>
<html>
   <head>
   <title> splice</title>
   <h1>  splice </h1>
</head>
<body bgcolor="pink">
<body>
   <p> splice  is which position we want to delete in the array element.</p>
   <script>
      var flowers = ["rose","jasmine","lilly","lotus"]; //its an array
      var removed = flowers.splice(1,1);// jasmine will removed
      document.write(flowers +"<br>");//print rose,lilly,lotus
      document .write(removed + "<br>");//prit:jasmine
      document.write(removed.length  + "<br>");//1

removed=flowers.splice( 2,1,"sunflower","tulip");
document.write(flowers + "<br>");//prints rose,lilly, sunflower,tulip
document.write(removed + "<br>");//print lotus
  document.write(flowers.length + "<br>")// print the length is 4
```

```
removed=flowers.splice(2,0,"marigold","daisy");
document.write(flowers + "<br>");// prints rose, lilly, marigold, daisy,sunflower,tulip
document.write(removed + "<br>");//removed is an empty

   </script>
</body>
</html>
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Strings</h1>
<h2>The length Property</h2>
<body bgcolor="pink";>
<p>The length of the string is:</p>
<p id="demo"></p>

<script>
let hotel =["ambriyani","barbiequeen","rasikas","nilasooru","kumaran"];

document.write(hotel.length);
</script>
<hr>
<h1>extracting string path</h1>
<h2>slice method</h2>
<script>
let foodreceipe = "tomatorice,curdrice,Lemonrice,tigerrice,mangorice";
let text= foodreceipe.slice(11,20);

document.write(text);
</script>
```

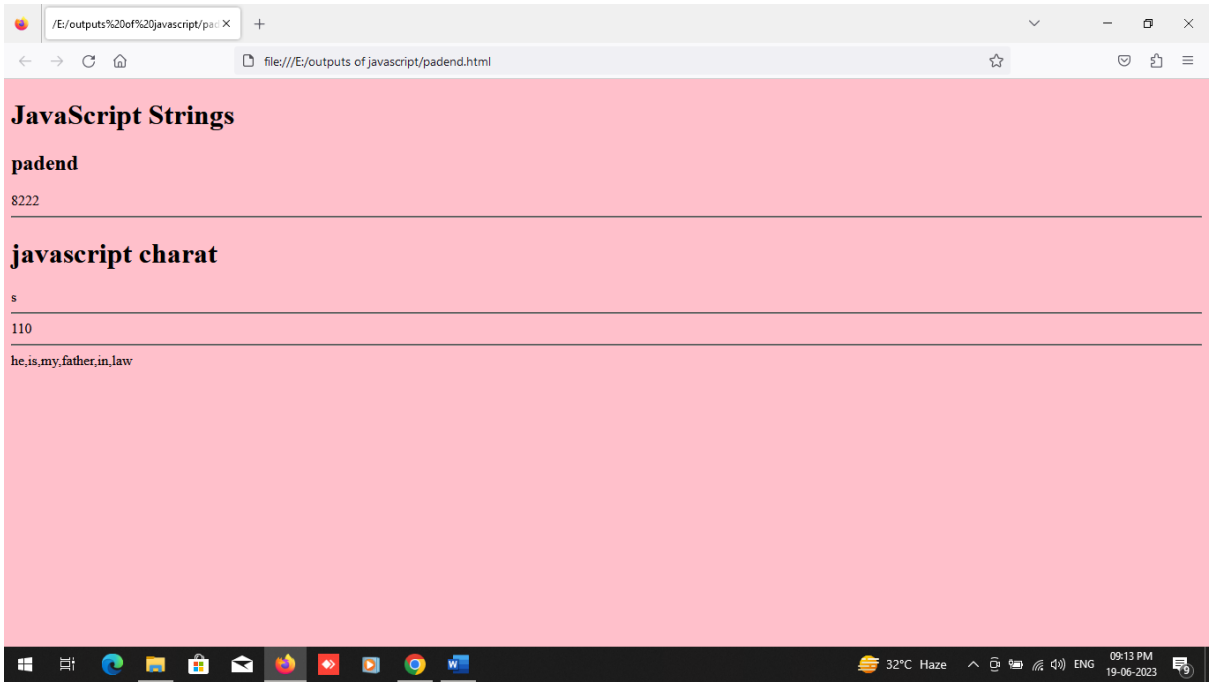
```
<hr>
<h1>JavaScript Strings</h1>
<h2>The substring() Method</h2>

<p>substring() extracts a part of a string:</p>
<script>
let sakthii = "wall clock time is cost effective!";
let fridge = sakthii.substring(20,10);

document.write(fridge);
</script>
<hr>
<h1>JavaScript Strings</h1>
<h2>The substr() Method</h2>

<p>substr() extracts a part of a string:</p>
<script>
var bag  = "sakthiikajol";
var element = bag.substr(2,9);

document.write(element);
</script>
<hr>
<h1>JavaScript Strings</h1>
<h2>The concat() Method</h2>

<p>The concat() method joins two or more strings.</p>

<p>The birds were singing outside our window. He's a tough old bird. We met some "smashing birds" at the pub "last night".:</p>
```

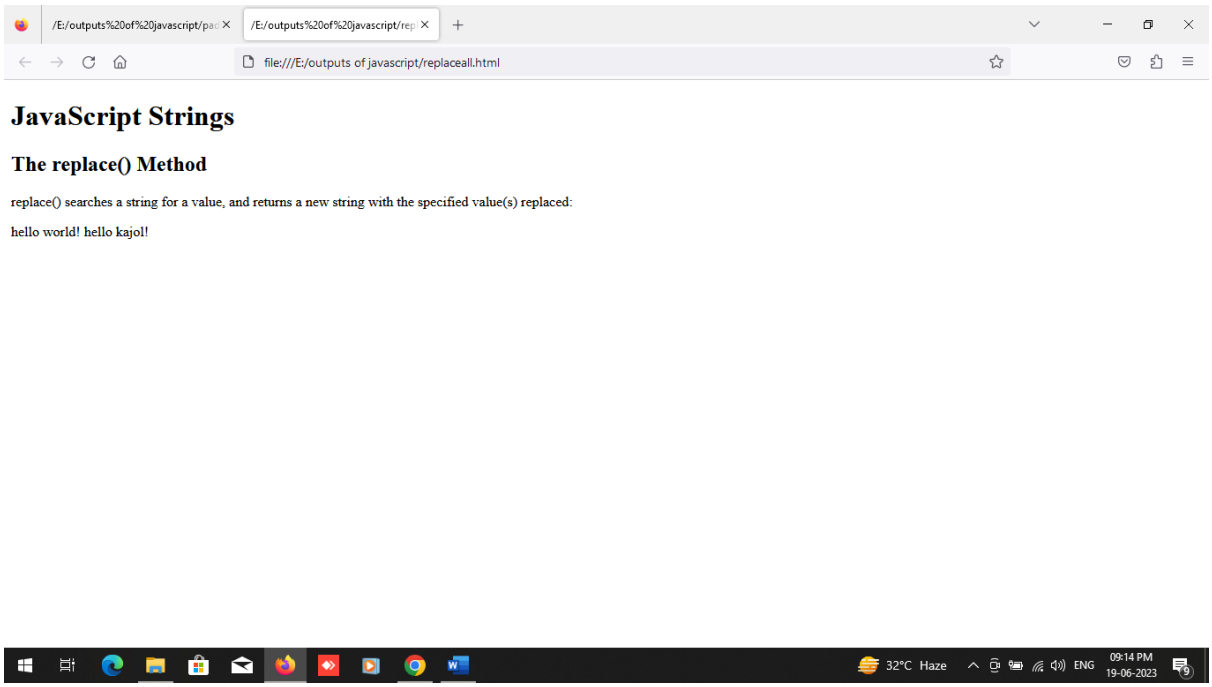
```
<p id="demo"></p>

<script>
var kajol = "smashing birds";
var sakthii = "last night";
var swetha = kajol.concat(sakthii);

document.getElementById("demo").innerHTML = swetha;
</script>

</body>

</html>
<!DOCTYPE html>
<html>
<body>

<h1>JavaScript Strings</h1>
<h2>The upper case </h2>
<body bgcolor="pink">
<script>
let furniture = "bench";
document.write(furniture.toUpperCase());
</script>
<hr>
<script>
var school = "STUDENT";
document.write(school.toLowerCase());
</script>
<hr>
<h1>javascript trim</h1>
```

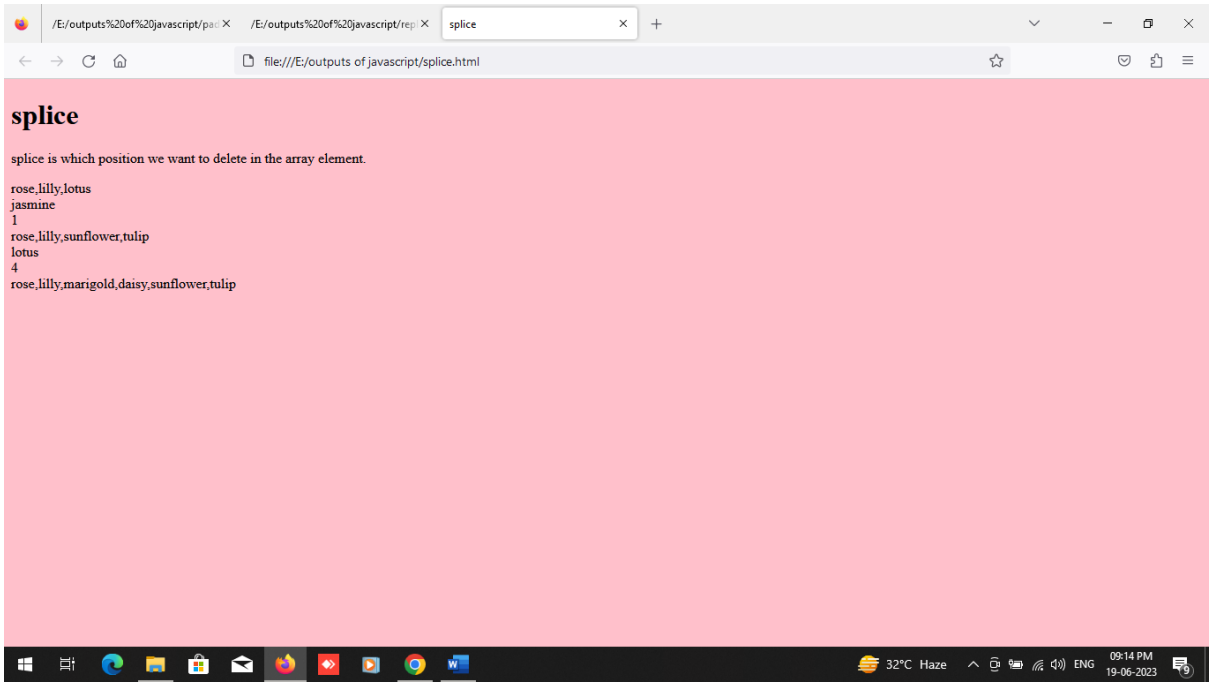
```
<script>
const animal = "                    elephant                    ";
let text = animal.trim();
document.write(text + "<br>");
document.write(animal.length + "<br>");
document.write(text.length);
</script>
<hr>
<script>
const animals = "                    elephant                    ";
let kajol = animals.trimStart();
document.write(kajol + "<br>");
document.write(animals.length + "<br>");
document.write(kajol.length);
</script>
<hr>
<script>
var cat = "                    elephant                    ";
let sakthii = cat.trimEnd();
document.write(sakthii + "<br>");
document.write(cat.length + "<br>");
document.write(sakthii.length);
</script>
<hr>
<script>
var cats = "8";

document.write(cats.padStart(7,"0") + "<br>");
</script>
<hr>
```

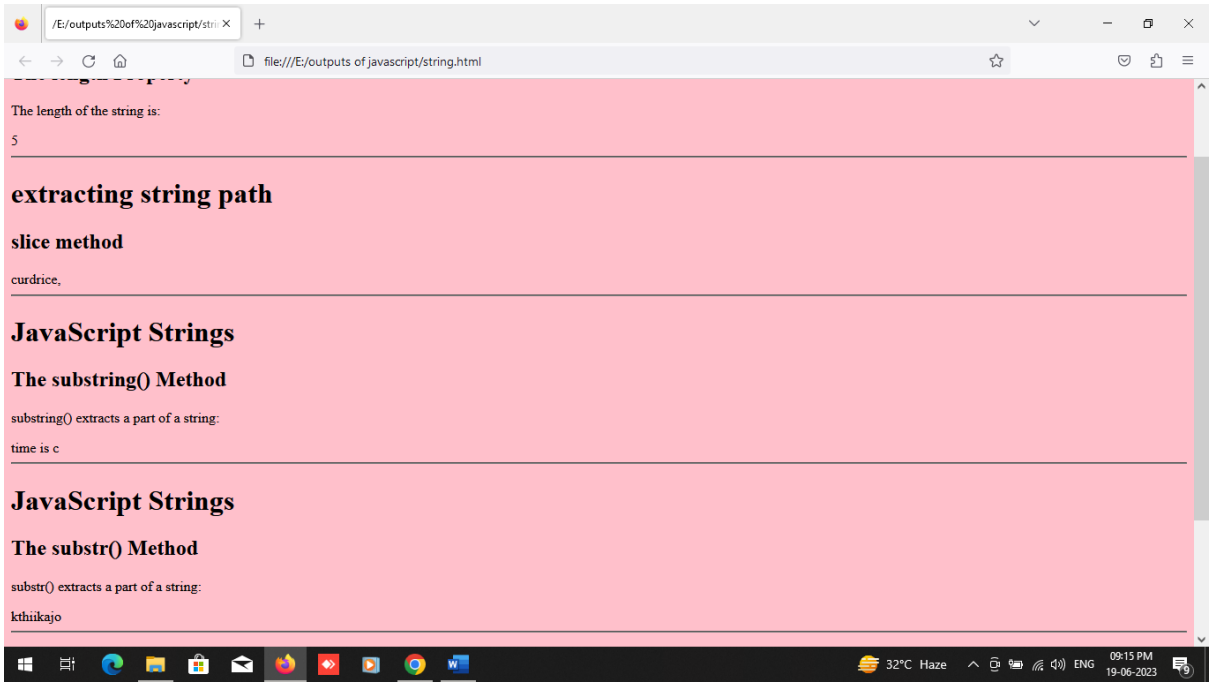
```
<script>
const baby = "8";
document.write(baby.padEnd(4,"3");
</script>
</body>
</html>
```

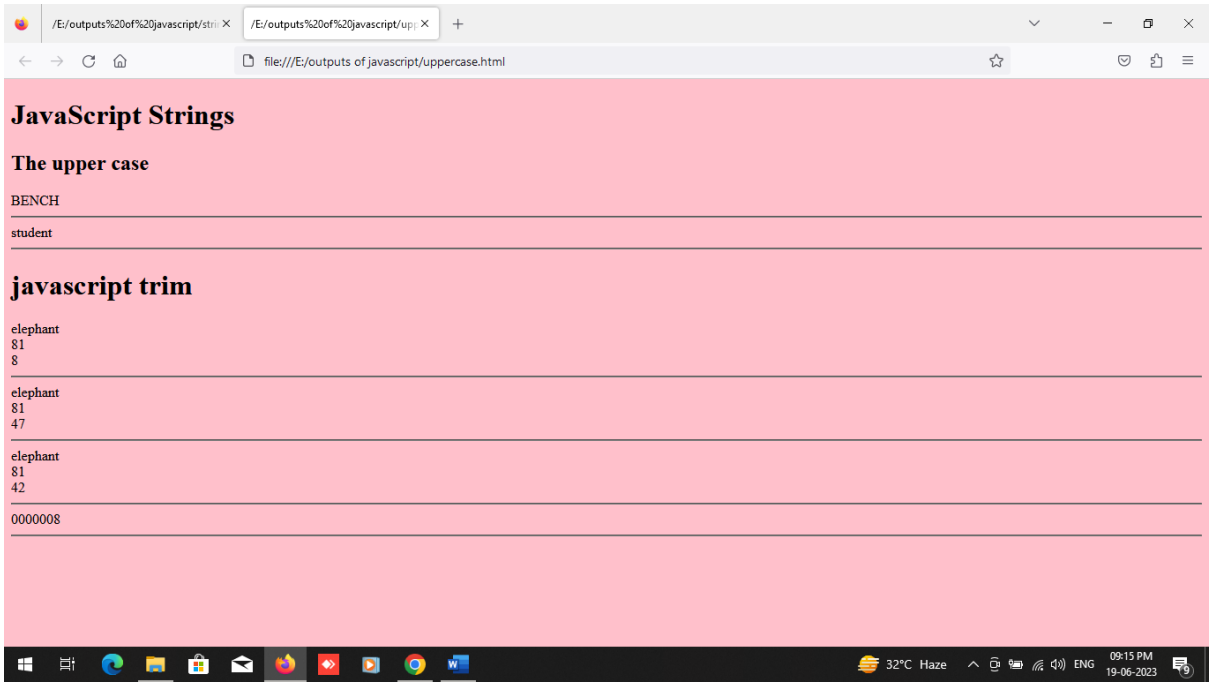
Output:






splice
file:///E:/outputs of javascript/splice.html
splice
splice is which position we want to delete in the array element.
rose,lilly,lotus
jasmine
1
rose,lilly,sunflower,tulip
lotus
4
rose,lilly,marigold,daisy,sunflower,tulip
32°C Haze
ENG
09:14 PM
19-06-2023


file:///E:/outputs of javascript/string.html
The length of the string is:
5
extracting string path
slice method
curdrice,
JavaScript Strings
The substring() Method
substring() extracts a part of a string:
time is c
JavaScript Strings
The substr() Method
substr() extracts a part of a string:
kthiikajo
32°C Haze
ENG
09:15 PM
19-06-2023



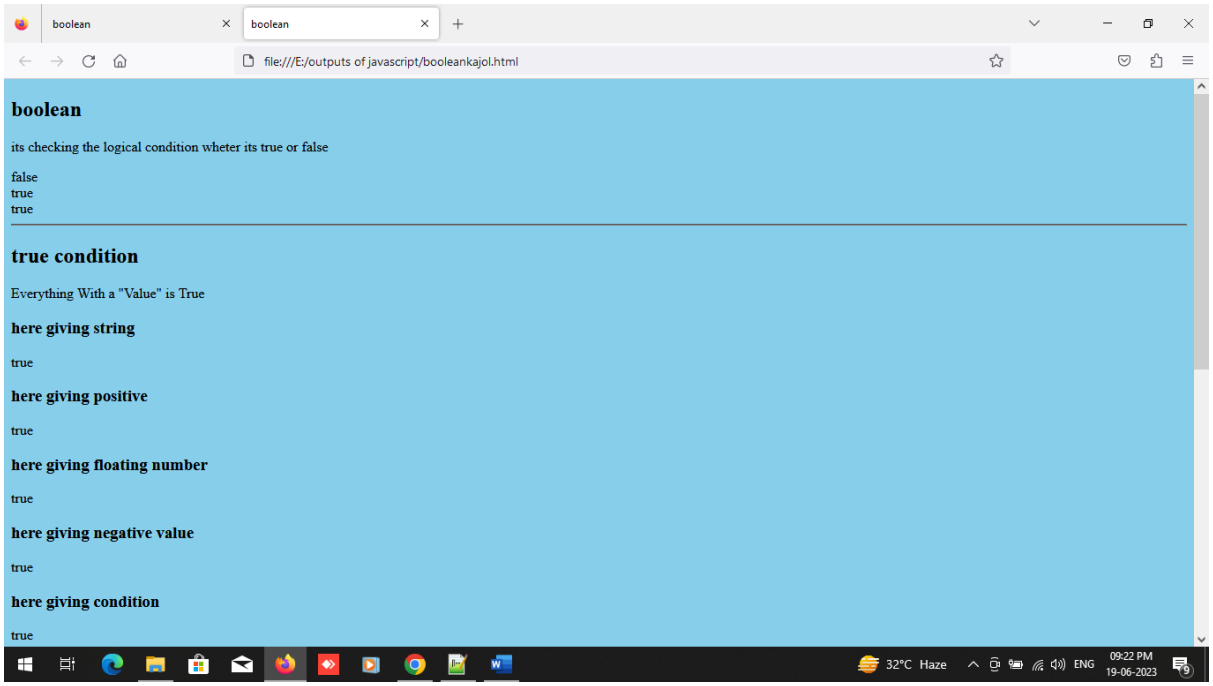

```
<!DOCTYPE html>
<html>
  <head>
    <title> boolean</title>
  </head>
  <body>
    <h2> boolean</h2>
                <body bgcolor="skyblue">
    <p> its checking the logical condition wheter its true or false</p>
    <script>
      document.write(10<9);//print false

        document.write("<br>");

      document .write(9>6);//print true
      document.write("<br>");
      document.write(0==0);
```

```
</script>
<hr>
   <h2> true condition</h2>
   <p>Everything With a "Value" is True</p>
   <h3>here  giving string </h3>
   <script>
       document.write( Boolean("kajol") + "<br>");
      </script>
      <h3>here  giving positive</h3>
      <script>
       document.write( Boolean("4") + "<br>");
      </script>
       <h3>here  giving floating number</h3>
       <script>
       document.write( Boolean("0.76") + "<br>");
      </script>
       <h3>here  giving negative value</h3>
       <script>
       document.write( Boolean("-4") + "<br>");
      </script>
       <h3>here  giving condition</h3>
       <script>
       document.write( Boolean("true") + "<br>");
      </script>
       <h3>here  giving mulitiplication </h3>
       <script>
       document.write( Boolean("3*6*5")+ "<br>");
   </script>
   <hr>
   <h2> false condition</h2>
   <p>Everything Without a "Value" is False</p>
```

```
        <h3>here giving zero"0"</h3>
        <script>

           let x = 0;
           document.write(Boolean(x));
        </script>
        <br>
        <h3>here  giving empty space</h3>
<script>
   let y="";
   document.write(Boolean(y));
</script>
<h3>undefined value</h3>
<script>
   var z;
   document.write(Boolean(z));
</script>
<h3>here give null value</h3>
<script>

   let s = null;
   document.write(Boolean(s));
</script>
<h3>here give NAN value</h3>
<script>

   let d = 10/ "ruthra";
   document.write(Boolean(d));
</script>
```

```
  </body>
</html>
```

Output:

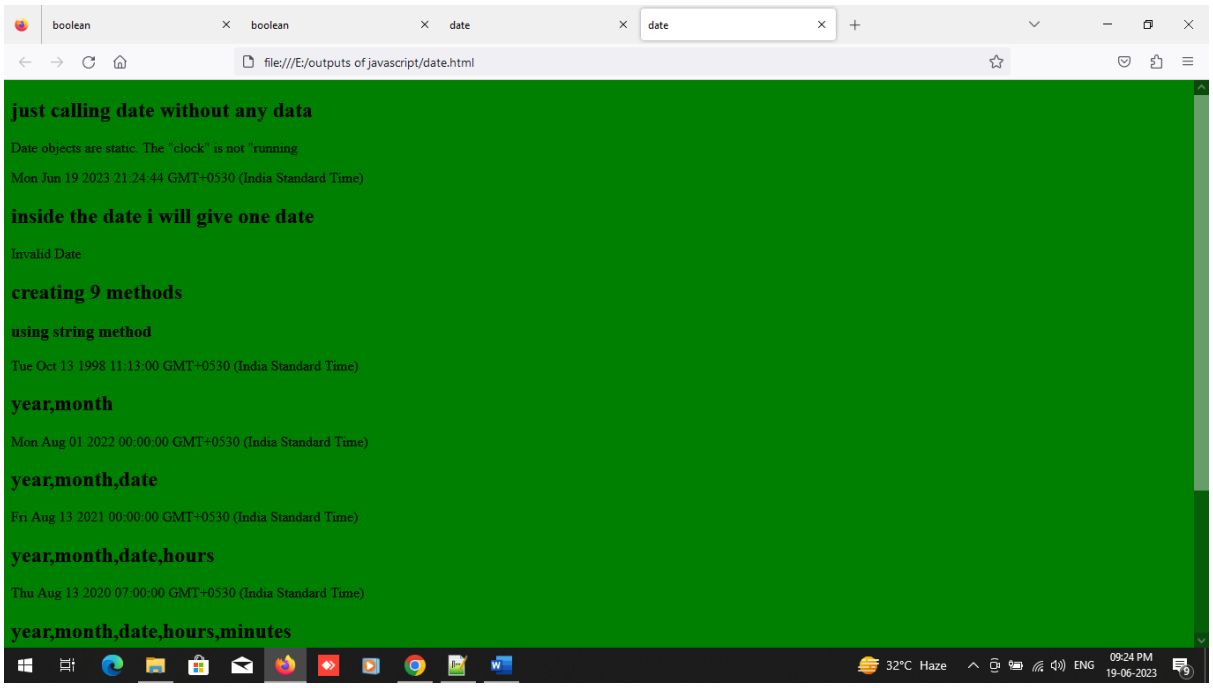



```
<!DOCTYPE html>
<html>
  <head>
    <title>date</title>
  </head>
  <body>
    <h2>  just calling date without any data</h2>
                <body bgcolor="green">
    <p>Date objects are static. The "clock" is not "running</p>
    <script>
      let x = new Date();
      document.write(x);

    </script>
    <h2> inside the date i will give one date</h2>
    <script>
```

```
    let y = new Date("2023-13-10");
    document.write(y);
</script>

<h2> creating 9 methods</h2>
<h3> using string method</h3>
<script>
    let z=new Date("oct 13, 1998 11:13:00")
    document.write(z);
</script>
<h2> year,month</h2>
    <script>
    let d3 = new Date(2022,07)
    document.write(d3);
</script>
<h2> year,month,date</h2>
    <script>
    let d2= new Date(2021, 7, 13)
    document.write(d2);
</script>
<h2> year,month,date,hours</h2>
    <script>
     let d1 = new Date(2020, 7, 13, 7)
    document.write(d1);
</script>
 <h2> year,month,date,hours,minutes</h2>
    <script>
    let c = new Date(2019, 7, 13, 7, 56)
    document.write(c);
</script>
<h2> year,month,date,hours,minutes,seconds</h2>
```

```
        <script>
         let b= new Date(2018, 4, 27, 4, 56, 67)
        document.write(b);
      </script>
      <h2> year,month,date,hours,minutes,seconds,milliseconds</h2>
        <script>
      let a = new Date(2016, 4, 27, 4, 56, 67, 45)
        document.write(a);
      </script>
   </body>
</html>
```

Output:

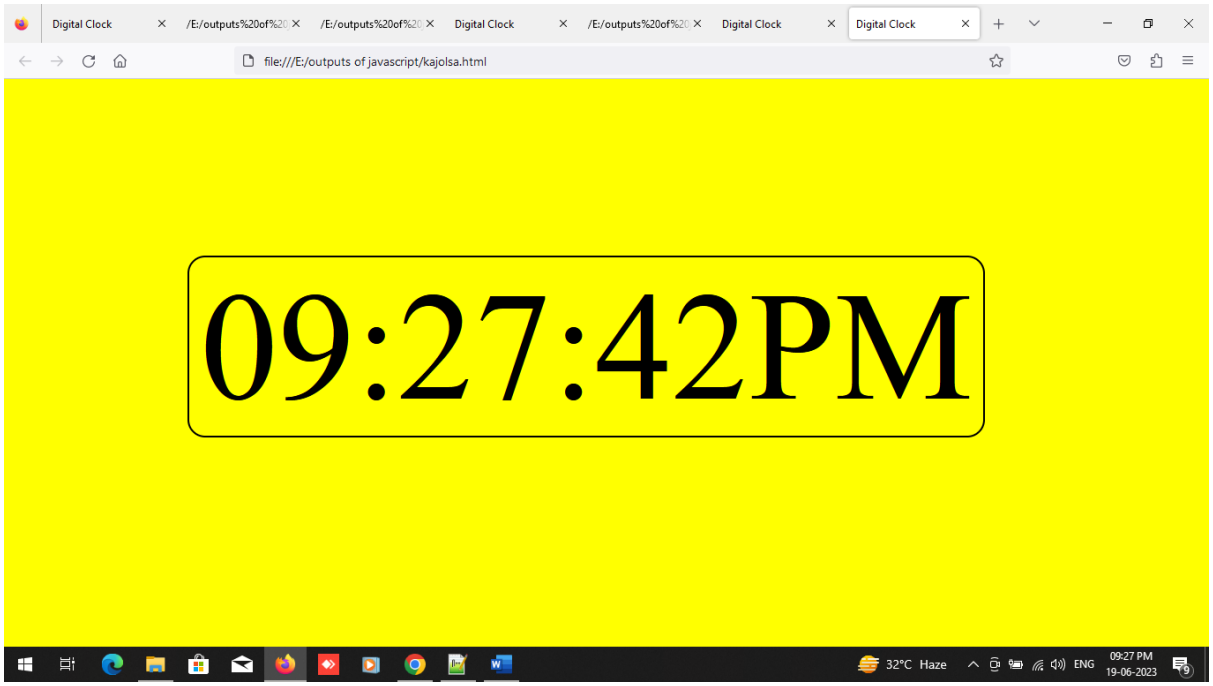



```
<!DOCTYPE html>
<html lang="en">

<head>
<body bgcolor="yellow">

   <meta charset="UTF-8" />
```

```
    <meta name="viewport" content="width=device-width,
                    initial-scale=1.0" />

    <title>Digital Clock</title>

    <style>
        #clock {
            font-size: 175px;
            width: 900px;
            margin: 200px;
            text-align: center;
            border: 2px solid black;
            border-radius: 20px;
        }
    </style>
</head>

<body>
    <div id="clock">8:10:45</div>

    <script>
        setInterval(showTime, 1000);
        function showTime() {
            let time = new Date();
            let hour = time.getHours();
            let min = time.getMinutes();
            let sec = time.getSeconds();
            am_pm = "AM";

            if (hour > 12) {
                    hour -= 12;
```

```
            am_pm = "PM";
        }
        if (hour == 0) {
            hr = 12;
            am_pm = "AM";
        }

        hour = hour < 10 ? "0" + hour : hour;
        min = min < 10 ? "0" + min : min;
        sec = sec < 10 ? "0" + sec : sec;

        let currentTime = hour + ":"
            + min + ":" + sec + am_pm;

        document.getElementById("clock")
            .innerHTML = currentTime;
    }

    showTime();
  </script>
</body>

</html>
```

Output:



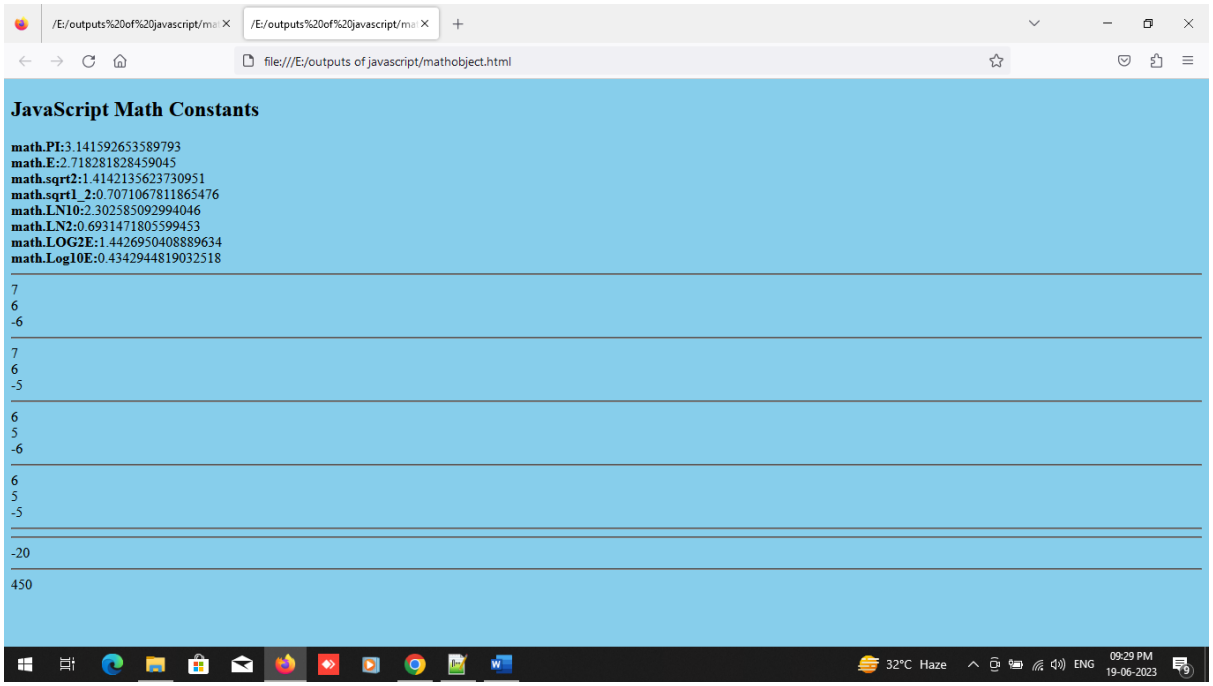

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Math Constants</h2>
<body bgcolor="skyblue">

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

<script>

        document.write("<b>math.PI:</b>" +Math.PI + "<br>");
        document.write("<b>math.E:</b>" + Math.E + "<br>");
        document.write("<b>math.sqrt2:</b>" + Math.SQRT2 + "<br>");
        document.write("<b>math.sqrt1_2:</b>" + Math.SQRT1_2 + "<br>");
        document.write("<b>math.LN10:</b>" + Math.LN10 + "<br>");
        document.write("<b>math.LN2:</b>" + Math.LN2+ "<br>");
        document.write("<b>math.LOG2E:</b>" + Math.LOG2E + "<br>");
```

```
document.write("<b>math.Log10E:</b>" + Math.LOG10E + "<br>");


</script>
                    <hr>
                    <script>
                    document.write(Math.round(6.5)+"<br>");
                    document.write(Math.round(5.6)+"<br>");
                    document.write(Math.round(-5.6)+"<br>");
                    </script>
                    <hr>
                    <script>
                    document.write(Math.ceil(6.5)+"<br>");
                    document.write(Math.ceil(5.6)+"<br>");
                    document.write(Math.ceil(-5.6)+"<br>");
                    </script>
                    <hr>
                    <script>
                    document.write(Math.floor(6.5)+"<br>");
                    document.write(Math.floor(5.6)+"<br>");
                    document.write(Math.floor(-5.6)+"<br>");
                    </script>
                    <hr>
                    <script>
                    document.write(Math.trunc(6.5)+"<br>");
                    document.write(Math.trunc(5.6)+"<br>");
                    document.write(Math.trunc(-5.6)+"<br>");
                    </script>
                    <hr>
                    <script>
                    let x=(100,150,0,10,,-80,-20);
                    document.write(Math.maxi(x)+"<br>");
```

```
                </script>
                <hr>
                <script>
                let x=(100, 150, 0, 10, -80, -20);
        document.write(Math.min(x) +"<br>");
                                </script>
                                <hr>
                <script>
                let b = Math.max(0, 450, 329, 87,  -46, 56);
        document.write(b +"<br>");
                                </script>
```

```
</body>
</html>
```

Output:

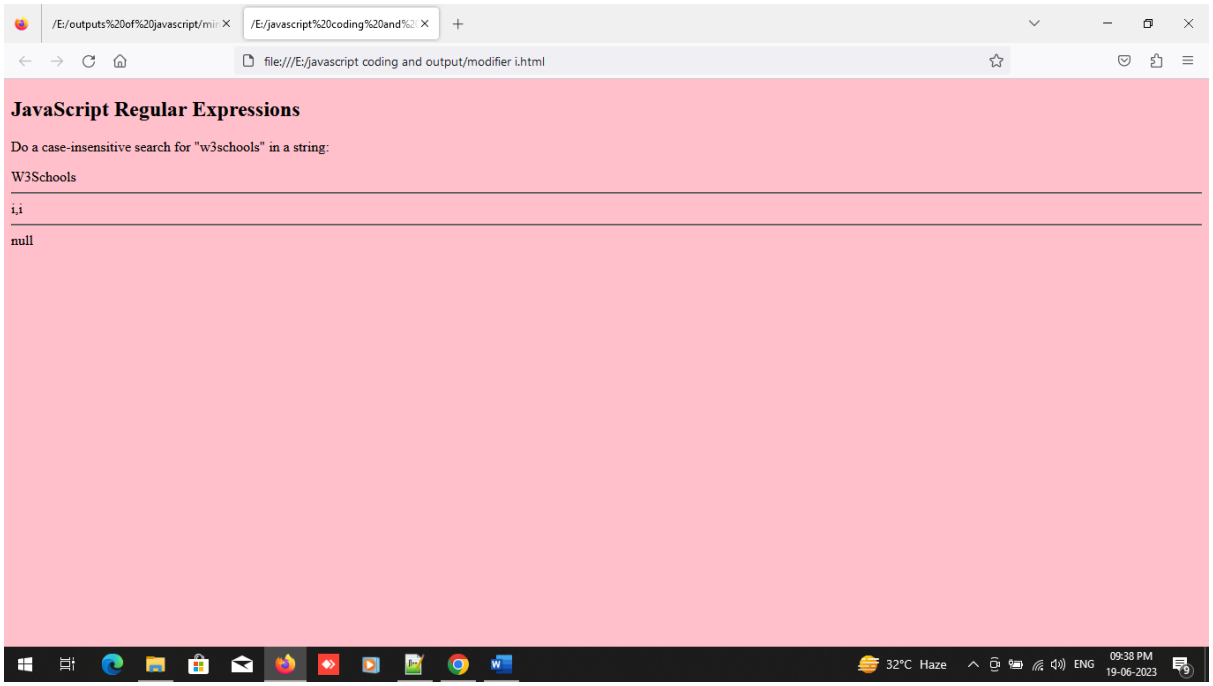


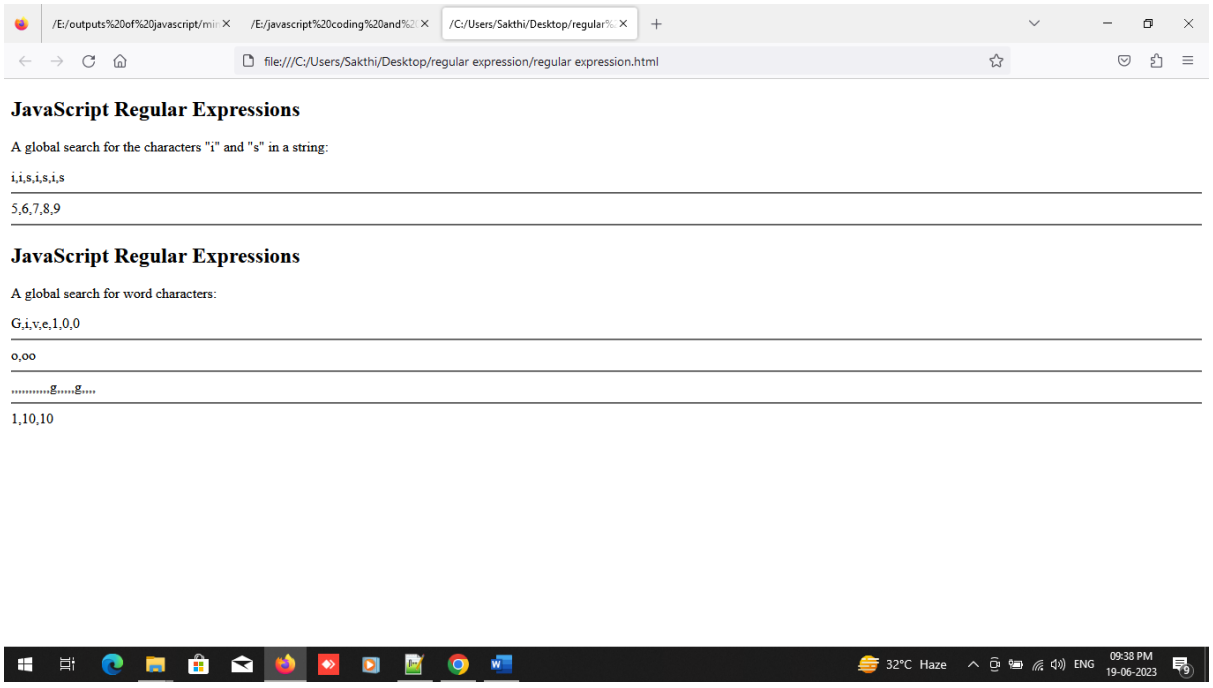
```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Regular Expressions</h2>
<body bgcolor="pink">

<p>Do a case-insensitive search for "w3schools" in a string:</p>




<script>
let text = "Visit W3Schools";
let pattern = /w3schools/i;
document.write(text.match(pattern) + "<br>");



</script>
<hr>
<script>
var name = "sakthii kajol";
var table =/i/g;
document.write(name.match(table) + "<br>");
</script>
<hr>
<script>
const animal ="lion is a king of forest";
const guru =/^f/m;
document.write(animal.match(guru) + "<br>");
</script>
</body>
```

```
</html>
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript Regular Expressions</h2>

<p>A global search for the characters "i" and "s" in a string:</p>

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

<script>
let text = "Do you know if this is all there is?";
let pattern = /[is]/gi;
document.write(text.match(pattern)  + "<br>");
</script>
<hr>
<script>
let x = "123456789";
let y = x.match( /[^1-4]/g);
document.write(y  + "<br>");
</script>
<hr>
<h2>JavaScript Regular Expressions</h2>

<p>A global search for word characters:</p>

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

<script>
let z = "Give 100%!";
```

```
let a = /\w/g;
document.write(z.match(a)  + "<br>");
</script>
<hr>
<script>
let b = "kajol is a good girl"
let c = /o+/g;
document.write(b.match(c)  + "<br>");
</script>
<hr>
<script>
let e = "kajol is a good girl"
let d = /g*/g;
document.write(e.match(d)  + "<br>");
</script>
<hr>
<script>
let f = "1, 100 or 1000?";
let g = /10?/g;
document.write(f.match(g)   + "<br>");
</script>
</body>
</html>
```

Output:





Levels of Testing:

In this section, we are going to understand the various **levels of software testing**.

As we learned in the earlier section of the software testing tutorial that testing any application or software, the test engineer needs to follow multiple testing techniques.

In order to detect an error, we will implement software testing; therefore, all the errors can be removed to find a product with more excellent quality.

What are the levels of Software Testing?

Testing levels are the procedure for finding the missing areas and avoiding overlapping and repetition between the development life cycle stages. We have already seen the various phases such as **Requirement collection, designing, coding testing, deployment, and maintenance** of SDLC (Software Development Life Cycle).

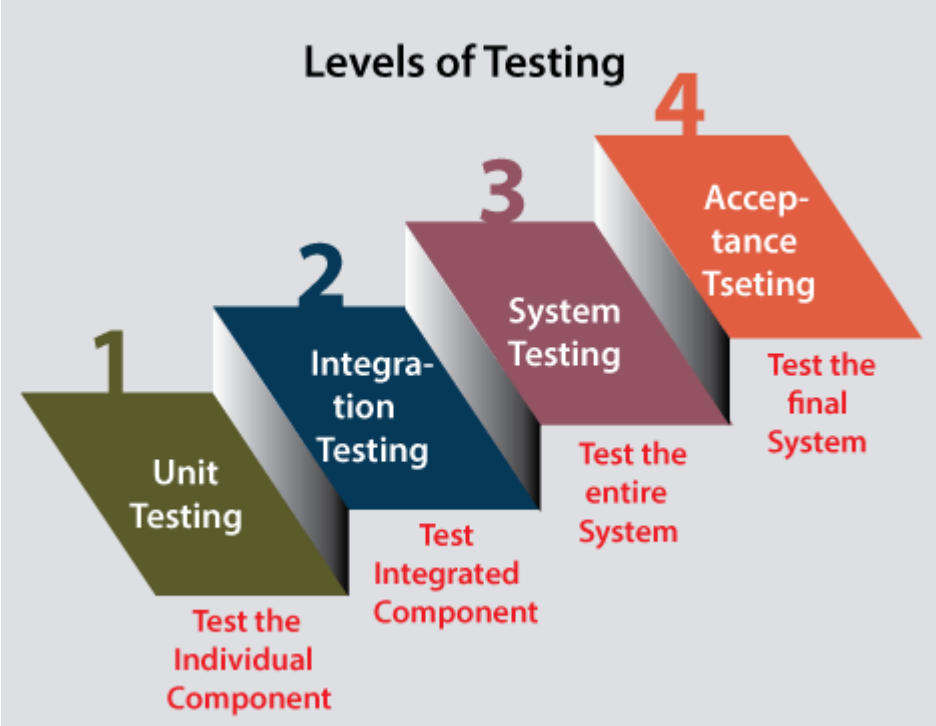
In order to test any application, we need to go through all the above phases of SDLC. Like SDLC, we have multiple levels of testing, which help us maintain the quality of the software.

Different Levels of Testing

The levels of software testing involve the different methodologies, which can be used while we are performing the software testing.

In software testing, we have four different levels of testing, which are as discussed below:

1. **Unit Testing**
2. **Integration Testing**
3. **System Testing**
4. **Acceptance Testing**



As we can see in the above image that all of these testing levels have a specific objective which specifies the value to the software development lifecycle.

For our better understanding, let's see them one by one:

Level1: Unit Testing

Unit testing is the first level of software testing, which is used to test if software modules are satisfying the given requirement or not.

The first level of testing involves **analyzing each unit or an individual component** of the software application.

Unit testing is also the first level of **functional testing**. The primary purpose of executing unit testing is to validate unit components with their performance.

A unit component is an individual function or regulation of the application, or we can say that it is the smallest testable part of the software. The reason of performing the unit testing is to test the correctness of inaccessible code.

Unit testing will help the test engineer and developers in order to understand the base of code that makes them able to change defect causing code quickly. The developers implement the unit.

Level2: Integration Testing

The second level of software testing is the **integration testing.** The integration testing process comes after **unit testing**.

It is mainly used to test the **data flow from one module or component to other modules.**

In integration testing, the **test engineer** tests the units or separate components or modules of the software in a group.

The primary purpose of executing the integration testing is to identify the defects at the interaction between integrated components or units.

When each component or module works separately, we need to check the data flow between the dependent modules, and this process is known as **integration testing**.

We only go for the integration testing when the functional testing has been completed successfully on each application module.

In simple words, we can say that **integration testing** aims to evaluate the accuracy of communication among all the modules.

Level3: System Testing

The third level of software testing is **system testing**, which is used to test the software's functional and non-functional requirements.

It is **end-to-end testing** where the testing environment is parallel to the production environment. In the third level of software testing, **we will test the application as a whole system.**

To check the end-to-end flow of an application or the software as a user is known as **System testing**.

In system testing, we will go through all the necessary modules of an application and test if the end features or the end business works fine, and test the product as a complete system.

The third level of software testing is **system testing**, which is used to test the software's functional and non-functional requirements.

It is **end-to-end testing** where the testing environment is parallel to the production environment. In the third level of software testing, **we will test the application as a whole system.**

To check the end-to-end flow of an application or the software as a user is known as **System testing**.

In system testing, we will go through all the necessary modules of an application and test if the end features or the end business works fine, and test the product as a complete system.

Level4: Acceptance Testing

The **last and fourth level** of software testing is **acceptance testing**, which is used to evaluate whether a specification or the requirements are met as per its delivery.

The software has passed through three testing levels (**Unit Testing, Integration Testing, System Testing**). Some minor errors can still be identified when the end-user uses the system in the actual scenario.

In simple words, we can say that Acceptance testing is the **squeezing of all the testing processes that are previously done.**

The acceptance testing is also known as **User acceptance testing (UAT)** and is done by the customer before accepting the final product.

Usually, UAT is done by the domain expert (customer) for their satisfaction and checks whether the application is working according to given business scenarios and real-time scenarios.

Conclusion

In this tutorial, we have learned all the levels of testing. And we can conclude that tests are grouped based on where they are added in the **Software development life cycle**.

A level of software testing is a process where every unit or component of a software or system is tested.

The main reason for implementing the **levels of testing** is to make the **software testing** process efficient and easy to find all possible test cases at a specific level.

To check the behavior or performance of software testing, we have various testing levels. The above-described software testing levels are developed to identify missing areas and understanding between the development life cycle conditions.

All these SDLC models' phases (**requirement gathering, analysis, design, coding or execution, testing, deployment, and maintenance**) undergo the process of software testing levels.

Functional and non-functional testing:

Functional Testing: Functional testing is a type of software testing in which the system is tested against the functional requirements and specifications. Functional testing ensures that the requirements or specifications are properly satisfied by the application. This type of testing is particularly concerned with the result of processing. It focuses on simulation of actual system usage but does not develop any system structure assumptions. It is basically defined as a type of testing which verifies that each function of the software application works in conformance with the requirement and specification. This testing is not concerned about the source code of the application. Each functionality of the software application is tested by providing appropriate test input, expecting the output and comparing the actual output with the expected output.
Non-functional Testing: Non-functional testing is a type of software testing that is performed to verify the non-functional

requirements of the application. It verifies whether the behavior of the system is as per the requirement or not. It tests all the aspects which are not tested in functional testing. Non-functional testing is defined as a type of software testing to check non-functional aspects of a software application. It is designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing. Non-functional testing is as important as functional testing.

They basically deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Difference between functional and non-functional testing:

Functional Testing	**Non-functional Testing**
It verifies the operations and actions of an application.	It verifies the behavior of an application.
It is based on requirements of customer.	It is based on expectations of customer.
It helps to enhance the behavior of the application.	It helps to improve the performance of the application.
Functional testing is easy to execute manually.	It is hard to execute non-functional testing manually.
It tests what the product does.	It describes how the product does.
Functional testing is based on the business requirement.	Non-functional testing is based on the performance requirement.
Examples: **1.** Unit Testing	**Examples:** **1.** Performance Testing

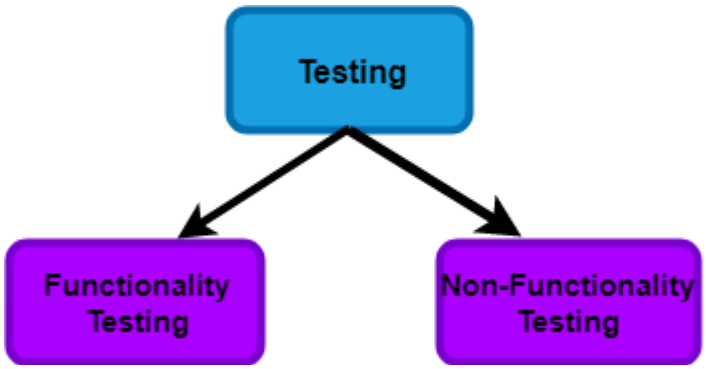
2. Smoke Testing **3.** Integration Testing **4.** Regression Testing	**2.** Load Testing **3.** Stress Testing **4.** Scalability Testing

Functional Testing

It is a type of software testing which is used to verify the functionality of the software application, whether the function is working according to the requirement specification. In functional testing, each function tested by giving the value, determining the output, and verifying the actual output with the expected value. Functional testing performed as black-box testing which is presented to confirm that the functionality of an application or system behaves as we are expecting. It is done to verify the functionality of the application.

Functional testing also called as black-box testing, because it focuses on application specification rather than actual code. Tester has to test only the program rather than the system.

There are two types of testing:

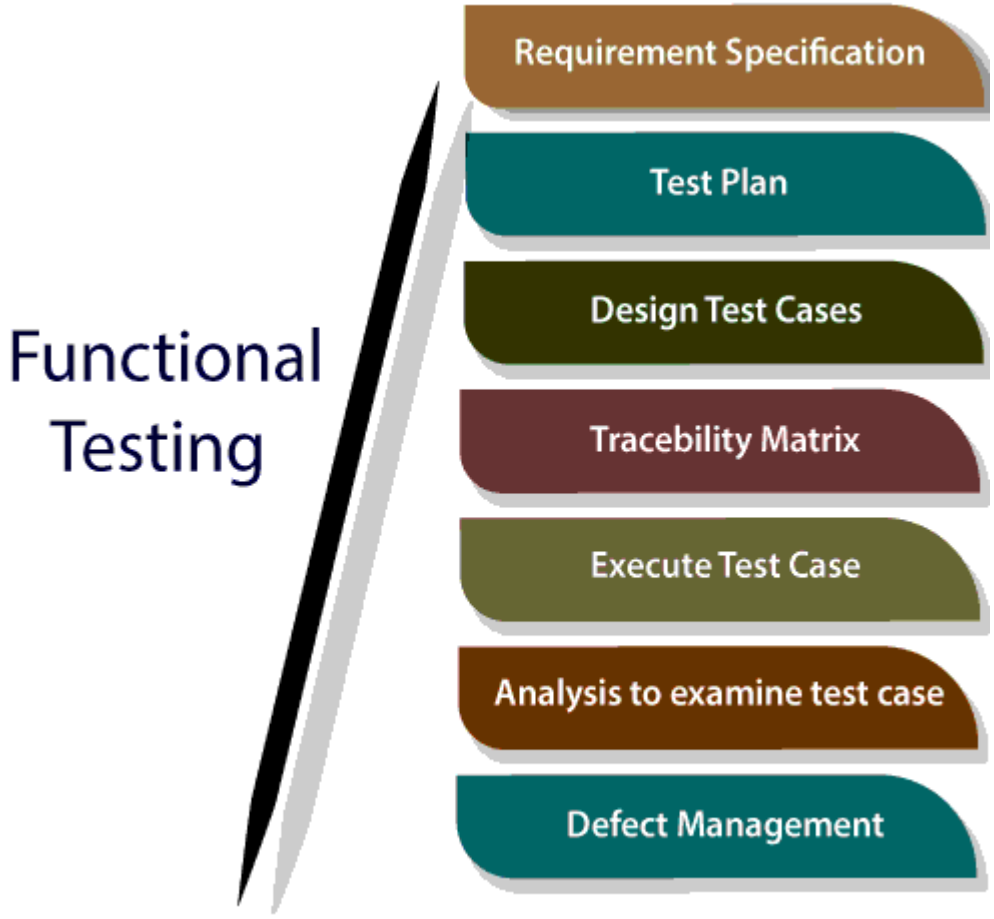


What is the process of functional testing?

Testers follow the following steps in the functional testing:

- Tester does verification of the requirement specification in the software application.
- After analysis, the requirement specification tester will make a plan.

- After planning the tests, the tester will design the test case.
- After designing the test, case tester will make a document of the traceability matrix.
- The tester will execute the test case design.
- Analysis of the coverage to examine the covered testing area of the application.
- Defect management should do to manage defect resolving.

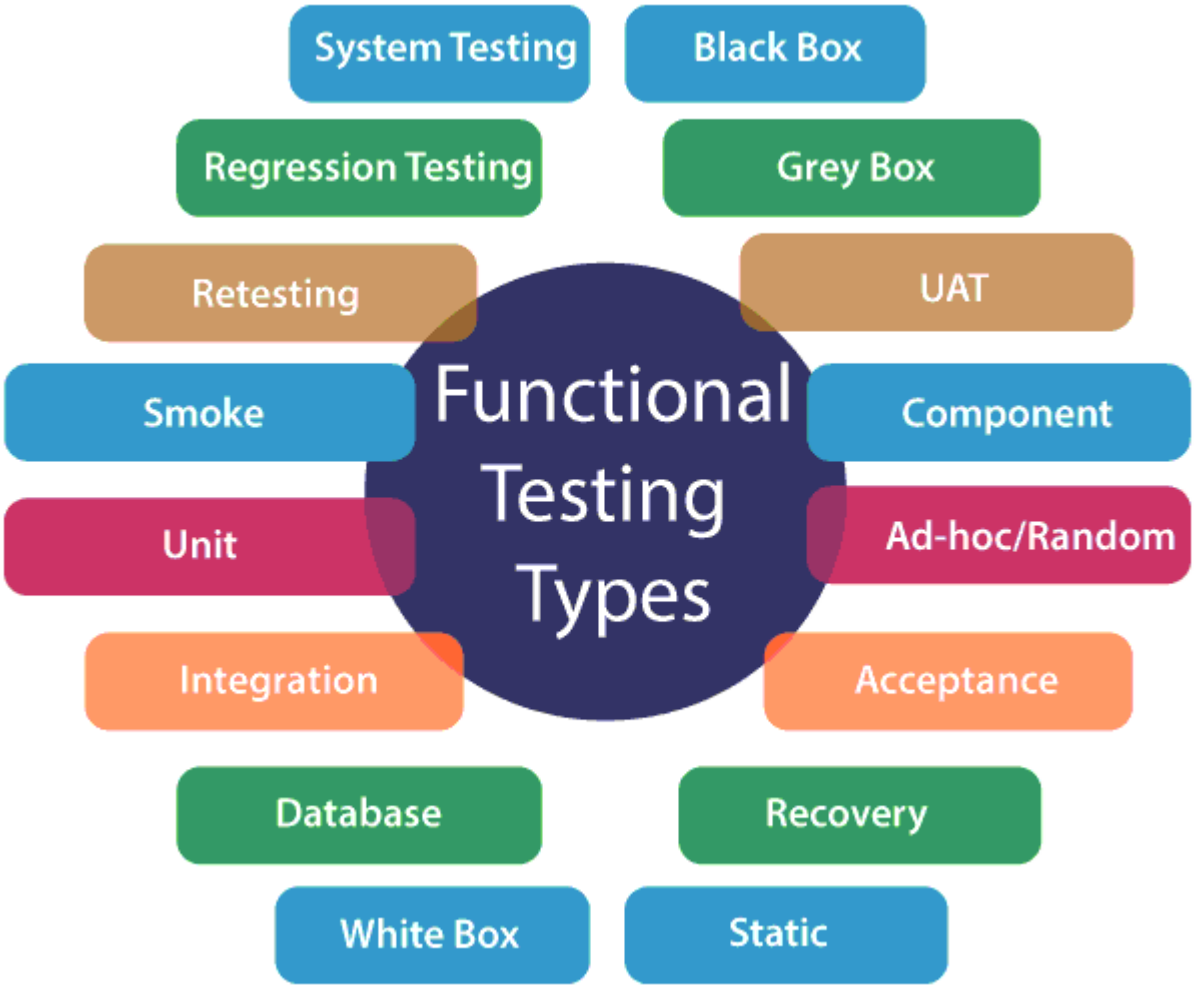


Explain the types of functional testing.

The main objective of functional testing is to test the functionality of the component.

Functional testing is divided into multiple parts.

Here are the following types of functional testing.



Unit Testing: Unit testing is a type of software testing, where the individual unit or component of the software tested. Unit testing, examine the different part of the application, by unit testing functional testing also done, because unit testing ensures each module is working correctly.

Smoke Testing: Functional testing by smoke testing. Smoke testing includes only the basic (feature) functionality of the system. Smoke testing is known as "***Build Verification Testing***." Smoke testing aims to ensure that the most important function work.

For example, Smoke testing verifies that the application launches successfully will check that GUI is responsive.

Sanity Testing: Sanity testing involves the entire high-level business scenario is working correctly. Sanity testing is done to check the

functionality/bugs fixed. Sanity testing is little advance than smoke testing.

For example, login is working fine; all the buttons are working correctly; after clicking on the button navigation of the page is done or not.

Regression Testing: This type of testing concentrate to make sure that the code changes should not side effect the existing functionality of the system. Regression testing specifies when bug arises in the system after fixing the bug, regression testing concentrate on that all parts are working or not. Regression testing focuses on is there any impact on the system.

Integration Testing: Integration testing combined individual units and tested as a group. The purpose of this testing is to expose the faults in the interaction between the integrated units.

Developers and testers perform integration testing.

White box testing: White box testing is known as Clear Box testing, code-based testing, structural testing, extensive testing, and glass box testing, transparent box testing. It is a software testing method in which the internal structure/design/ implementation tested known to the tester.

The white box testing needs the analysis of the internal structure of the component or system.

Black box testing: It is also known as behavioral testing. In this testing, the internal structure/ design/ implementation not known to the tester. This type of testing is functional testing. Why we called this type of testing is black-box testing, in this testing tester, can't see the internal code.

For example, A tester without the knowledge of the internal structures of a website tests the web pages by using the web browser providing input and verifying the output against the expected outcome.

User acceptance testing: It is a type of testing performed by the client to certify the system according to requirement. The final phase of

testing is user acceptance testing before releasing the software to the market or production environment. UAT is a kind of black-box testing where two or more end-users will involve.

Static testing helps to find the error in the early stages. With the help of static testing, this will reduce the development timescales. It reduces the testing cost and time. Static testing also used for development productivity.

Component Testing: Component Testing is also a type of software testing in which testing is performed on each component separately without integrating with other parts. Component testing is also a type of black-box testing. Component testing also referred to as Unit testing, program testing, or module testing.

Grey Box Testing: Grey Box Testing defined as a combination of both white box and black-box testing. Grey Box testing is a testing technique which performed with limited information about the internal functionality of the system.

Advantages of functional testing are:

- It produces a defect-free product.
- It ensures that the customer is satisfied.
- It ensures that all requirements met.
- It ensures the proper working of all the functionality of an application/software/product.
- It ensures that the software/ product work as expected.
- It ensures security and safety.
- It improves the quality of the product.

Disadvantages of functional testing are:

- Functional testing can miss a critical and logical error in the system.
- This testing is not a guarantee of the software to go live.
- The possibility of conducting redundant testing is high in functional testing.

Non-Functional Testing

Non-functional testing is a type of software testing to test non-functional parameters such as reliability, load test, performance and accountability of the software. The primary purpose of non-functional testing is to test the reading speed of the software system as per non-functional parameters. The parameters of non-functional testing are never tested before the functional testing.

Non-functional testing is also very important as functional testing because it plays a crucial role in customer satisfaction.

For example, non-functional testing would be to test how many people can work simultaneously on any software.

Parameters to be tested under Non-Functional Testing

Performance Testing

Performance Testing eliminates the reason behind the slow and limited performance of the software. Reading speed of the software should be as fast as possible.

For Performance Testing, a well-structured and clear specification about expected speed must be defined. Otherwise, the outcome of the test (Success or Failure) will not be obvious.

Load Testing

Load testing involves testing the system's loading capacity. Loading capacity means more and more people can work on the system simultaneously.

Security Testing

Security testing is used to detect the security flaws of the software application. The testing is done via investigating system architecture

and the mindset of an attacker. Test cases are conducted by finding areas of code where an attack is most likely to happen.

Portability Testing

The portability testing of the software is used to verify whether the system can run on different operating systems without occurring any bug. This test also tests the working of software when there is a same operating system but different hardware.

Accountability Testing

Accountability test is done to check whether the system is operating correctly or not. A function should give the same result for which it has been created. If the system gives expected output, it gets passed in the test otherwise failed.

Reliability Testing

Reliability test assumes that whether the software system is running without fail under specified conditions or not. The system must be run for a specific time and number of processes. If the system is failed under these specified conditions, reliability test will be failed.

Efficiency Testing

Efficiency test examines the number of resources needed to develop a software system, and how many of these were used. It also includes the test of these three points.

- Customer's requirements must be satisfied by the software system.
- A software system should achieve customer specifications.
- Enough efforts should be made to develop a software system.

Advantages of Non-functional testing

- It provides a higher level of security. Security is a fundamental feature due to which system is protected from cyber-attacks.

- It ensures the loading capability of the system so that any number of users can use it simultaneously.
- It improves the performance of the system.
- Test cases are never changed so do not need to write them more than once.
- Overall time consumption is less as compared to other testing processes.

Disadvantages of Non-Functional Testing

- Every time the software is updated, non-functional tests are performed again.
- Due to software updates, people have to pay to re-examine the software; thus software becomes very expensive.

White Box Testing

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**. It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing. In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software.

The term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Developers do white box testing. In this, the developer will test every line of the code of the program. The developers perform the White-box testing and then send the application or the software to the testing team, where they will perform the black box testing and verify the application along with the requirements and identify the bugs and sends it to the developer.

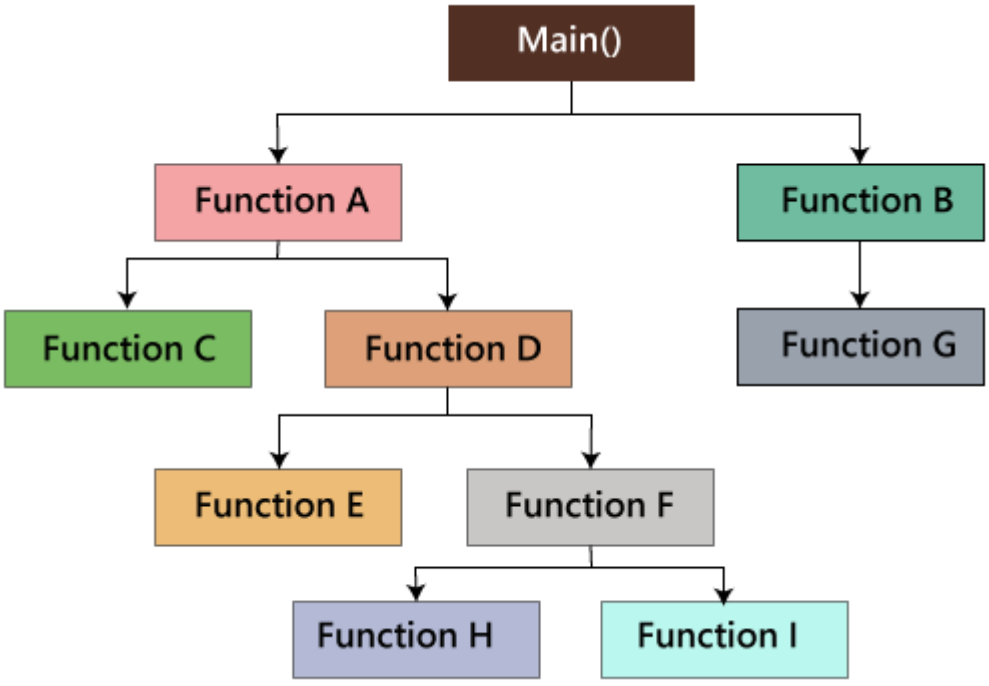
The developer fixes the bugs and does one round of white box testing and sends it to the testing team. Here, fixing the bugs implies that the bug is deleted, and the particular feature is working fine on the application.

The white box testing contains various tests, which are as follows:

- Path testing
- Loop testing
- Condition testing
- Testing based on the memory perspective
- Test performance of the program

Path testing

In the path testing, we will write the flow graphs and test all independent paths. Here writing the flow graph implies that flow graphs are representing the flow of the program and also show how every program is added with one another as we can see in the below image:



Loop testing

In the loop testing, we will test the loops such as while, for, and do-while, etc. and also check for ending condition if working correctly and if the size of the conditions is enough.

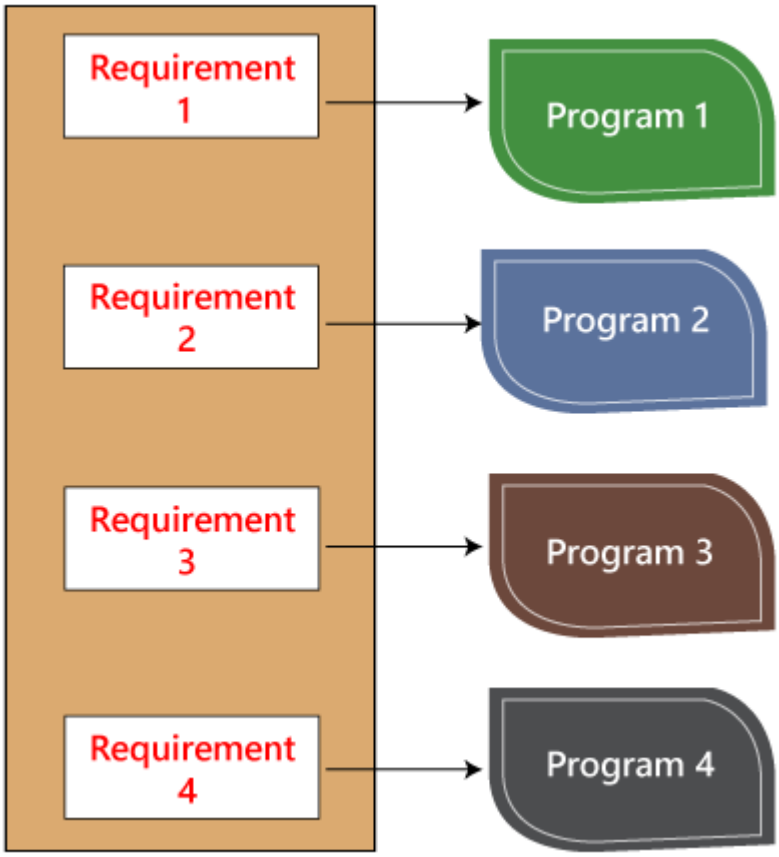
For example: we have one program where the developers have given about 50,000 loops.

1. {
2. while(50,000)
3. ……
4. ……
5. }

We cannot test this program manually for all the 50,000 loops cycle. So we write a small program that helps for all 50,000 cycles, as we can see in the below program, that test P is written in the similar language as the source code program, and this is known as a Unit test. And it is written by the developers only.

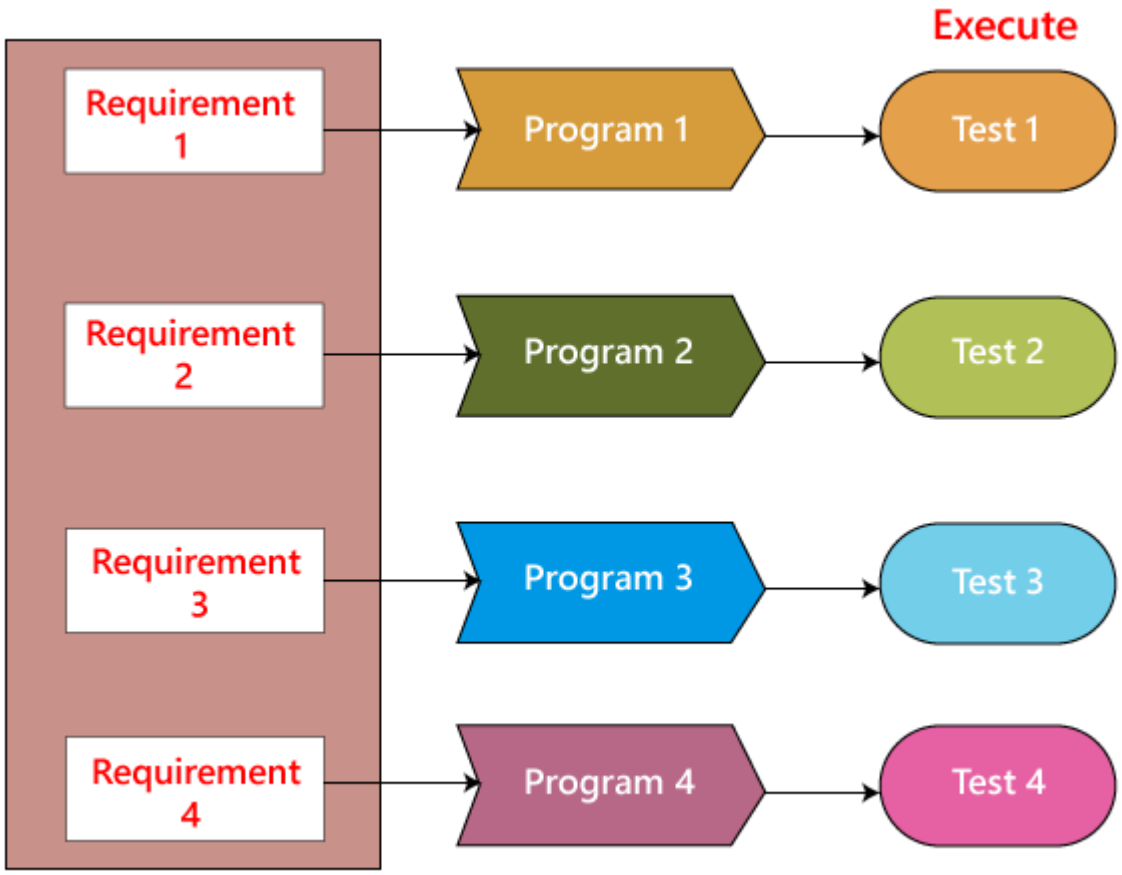
1. Test P
2. {
3. ……
4. …… }

As we can see in the below image that, we have various requirements such as 1, 2, 3, 4. And then, the developer writes the programs such as program 1,2,3,4 for the parallel conditions. Here the application contains the 100s line of codes.



These issues can be resolved in the following ways:

In this, we will write test for a similar program where the developer writes these test code in the related language as the source code. Then they execute these test code, which is also known as **unit test programs**. These test programs linked to the main program and implemented as programs.



Therefore, if there is any requirement of modification or bug in the code, then the developer makes the adjustment both in the main program and the test program and then executes the test program.

Condition testing

In this, we will test all logical conditions for both **true** and **false** values; that is, we will verify for both **if** and **else** condition.

For example:

1. if(condition) - true
2. {
3. …..
4. ……
5. ……
6. }
7. else - false
8. {

9. …..
10. ……
11. ……
12. }

The above program will work fine for both the conditions, which means that if the condition is accurate, and then else should be false and conversely.

Testing based on the memory (size) perspective

The size of the code is increasing for the following reasons:

- **The reuse of code is not there**: let us take one example, where we have four programs of the same application, and the first ten lines of the program are similar. We can write these ten lines as a discrete function, and it should be accessible by the above four programs as well. And also, if any bug is there, we can modify the line of code in the function rather than the entire code.
- The **developers use the logic** that might be modified. If one programmer writes code and the file size is up to 250kb, then another programmer could write a similar code using the different logic, and the file size is up to 100kb.
- The **developer declares so many functions and variables** that might never be used in any portion of the code. Therefore, the size of the program will increase.

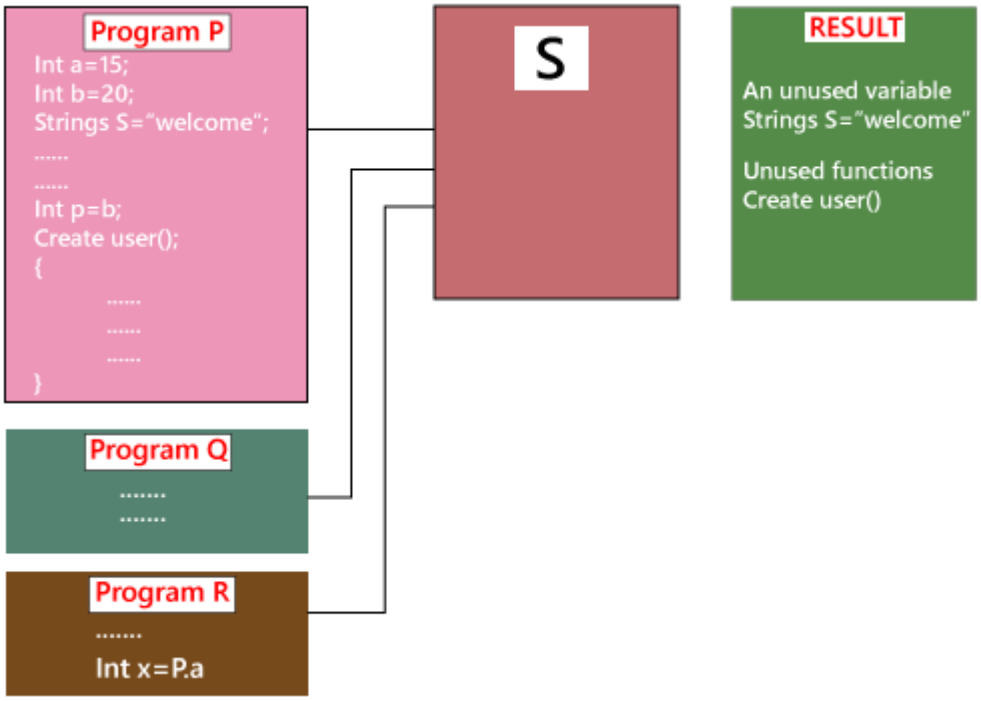
For example,

1. Int a=15;
2. Int b=20;
3. String S= "Welcome";
4. ….
5. …..
6. …..
7. ….
8. …..
9. Int p=b;

```
10.     Create user()
11.     {
12.     ……
13.     ……
14.     ….. 200's line of code
15.     }
```

In the above code, we can see that the **integer a** has never been called anywhere in the program, and also the function **Create user** has never been called anywhere in the code. Therefore, it leads us to memory consumption.

We cannot remember this type of mistake manually by verifying the code because of the large code. So, we have a built-in tool, which helps us to test the needless variables and functions. And, here we have the tool called **Rational purify**.



Suppose we have three programs such as Program P, Q, and R, which provides the input to S. And S goes into the programs and verifies the unused variables and then gives the outcome. After that, the developers will click on several results and call or remove the unnecessary function and the variables.

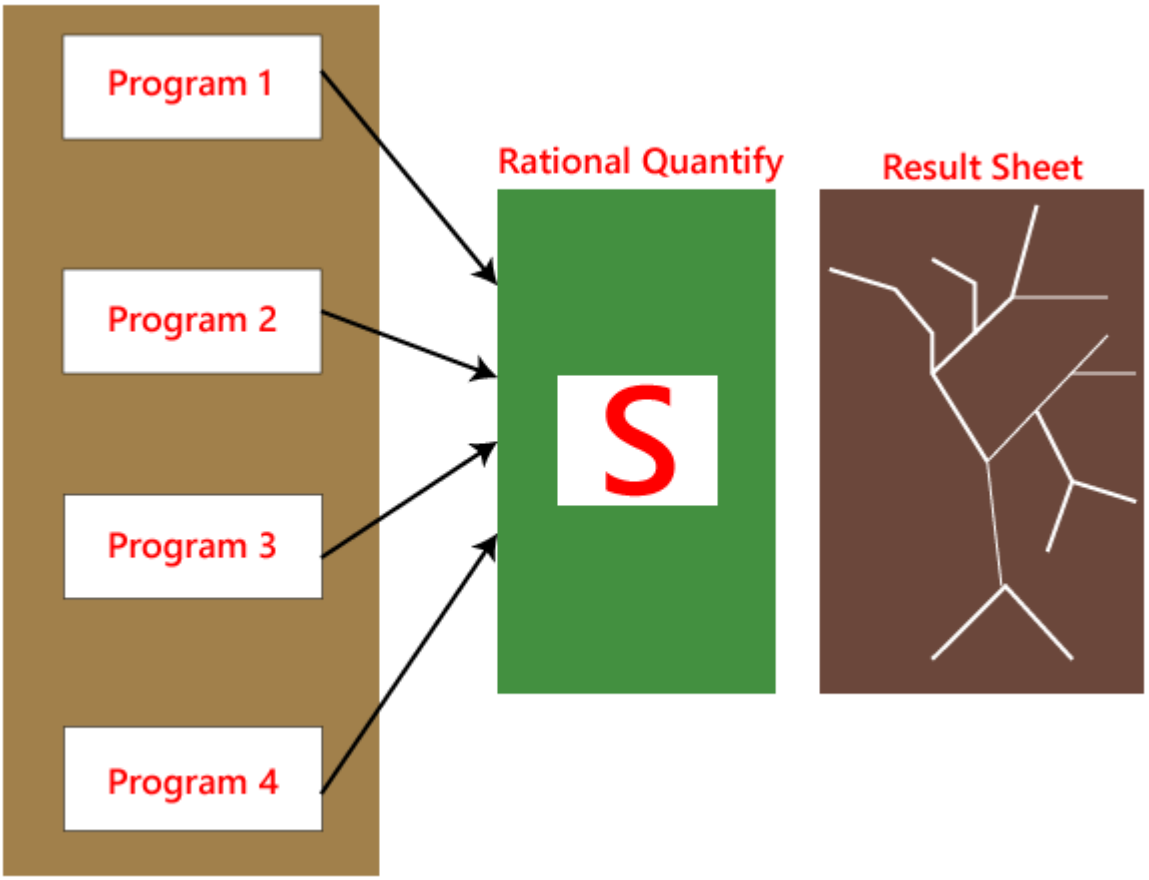
This tool is only used for the C programming language and C++ programming language; for another language, we have other related tools available in the market.

- The developer does not use the available in-built functions; instead they write the full features using their logic. Therefore, it leads us to waste of time and also postpone the product releases.

Test the performance (Speed, response time) of the program

The application could be slow for the following reasons:

- When logic is used.
- For the conditional cases, we will use **or** & **and** adequately.
- Switch case, which means we cannot use **nested if**, instead of using a switch case.

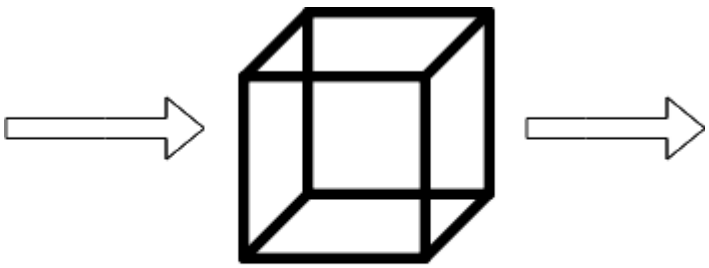


As we know that the developer is performing white box testing, they understand that the code is running slow, or the performance of the program is also getting deliberate. And the developer cannot go manually over the program and verify which line of the code is slowing the program.

To recover with this condition, we have a tool called **Rational Quantify**, which resolves these kinds of issues automatically. Once the entire code is ready, the rational quantify tool will go through the code and execute it. And we can see the outcome in the result sheet in the form of thick and thin lines.

Here, the thick line specifies which section of code is time-consuming. When we double-click on the thick line, the tool will take us to that line or piece of code automatically, which is also displayed in a different color. We can change that code and again and use this tool. When the order of lines is all thin, we know that the presentation of the program has enhanced. And the developers will perform the white box testing automatically because it saves time rather than performing manually.

Test cases for white box testing are derived from the design phase of the software development lifecycle. Data flow testing, control flow testing, path testing, branch testing, statement and decision coverage all these techniques used by white box testing as a guideline to create an error-free software.



Whitebox Testing

White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Generic steps of white box testing

- Design all test scenarios, test cases and prioritize them according to high priority number.
- This step involves the study of code at runtime to examine the resource utilization, not accessed areas of the code, time taken by various methods and operations and so on.
- In this step testing of internal subroutines takes place. Internal subroutines such as nonpublic methods, interfaces are able to handle all types of data appropriately or not.
- This step focuses on testing of control statements like loops and conditional statements to check the efficiency and accuracy for different data inputs.
- In the last step white box testing includes security testing to check all possible security loopholes by looking at how the code handles security.

Reasons for white box testing

- It identifies internal security holes.
- To check the way of input inside the code.
- Check the functionality of conditional loops.
- To test function, object, and statement at an individual level.

Advantages of White box testing

- White box testing optimizes code so hidden errors can be identified.
- Test cases of white box testing can be easily automated.
- This testing is more thorough than other testing approaches as it covers all code paths.
- It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

- White box testing is too much time consuming when it comes to large-scale programming applications.
- White box testing is much expensive and complex.

- It can lead to production error because it is not detailed by the developers.
- White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation.

Techniques Used in White Box Testing

Data Flow Testing	Data flow testing is a group of testing strategies that examines the control flow of programs in order to explore the sequence of variables according to the sequence of events.
Control Flow Testing	Control flow testing determines the execution order of statements or instructions of the program through a control structure. The control structure of a program is used to develop a test case for the program. In this technique, a particular part of a large program is selected by the tester to set the testing path. Test cases represented by the control graph of the program.
Branch Testing	Branch coverage technique is used to cover all branches of the control flow graph. It covers all the possible outcomes (true and false) of each condition of decision point at least once.
Statement Testing	Statement coverage technique is used to design white box test cases. This technique involves execution of all statements of the source code at least once. It is used to calculate the total number of executed statements in the source code, out of total statements present in the source code.
Decision Testing	This technique reports true and false outcomes of Boolean expressions. Whenever there is a possibility of two or more outcomes from the statements like do while statement, if statement and case statement (Control flow statements), it is considered as decision point because there are two outcomes either true or false.

Data Flow Testing

Data flow testing is used to analyze the flow of data in the program. It is the process of collecting information about how the variables flow the data in the program. It tries to obtain particular information of each particular point in the process.

Data flow testing is a group of testing strategies to examine the control flow of programs in order to explore the sequence of variables according to the sequence of events. It mainly focuses on the points at which values assigned to the variables and the point at which these values are used by concentrating on both points, data flow can be tested.

Let's understand this with an example:

1. read x;	
2. If(x>0)	(1, (2, t), x), (1, (2, f), x)
3. a= x+1	(1, 3, x)
4. if (x<=0) {	(1, (4, t), x), (1, (4, f), x)
5. if (x<1)	(1, (5, t), x), (1, (5, f), x)
6. x=x+1; (go to 5)	(1, 6, x)
else	
7. a=x+1	(1, 7, x)
8. print a;	(6,(5, f)x), (6,(5,t)x)
	(6, 6, x)
	(3, 8, a), (7, 8, a).

So, we are taking two paths to cover all the statements.

1. **x= 1**

Path - 1, 2, 3, 8

Output = 2

When we set value of x as 1 first it come on step 1 to read and assign the value of x (we took 1 in path) then come on statement 2 (x>0 (we took 2 in path)) which is true and it comes on statement 3 (a= x+1 (we took 3 in path)) at last it comes on statement 8 to print the value of x (output is 2).

For the second path, we take the value of x is 1

2. **Set x= -1**

Path = 1, 2, 4, 5, 6, 5, 6, 5, 7, 8

Output = 2

Control Flow Testing

Control flow testing is a testing technique that comes under white box testing. The aim of this technique is to determine the execution order of statements or instructions of the program through a control structure. The control structure of a program is used to develop a test case for the program. In this technique, a particular part of a large program is selected by the tester to set the testing path. It is mostly used in unit testing. Test cases represented by the control graph of the program.

Control Flow Graph is formed from the node, edge, decision node, junction node to specify all possible execution path.

Notations used for Control Flow Graph

1. Node
2. Edge
3. Decision Node
4. Junction node

Node

Nodes in the control flow graph are used to create a path of procedures. Basically, it represents the sequence of procedures which procedure is next to come so, the tester can determine the sequence of occurrence of procedures.

Edge

Edge in control flow graph is used to link the direction of nodes.

We can see below in example all arrows are used to link the nodes in an appropriate direction.

Decision node

Decision node in the control flow graph is used to decide next node of procedure as per the value.

We can see below in example decision node decide next node of procedure as per the value of n if it is 18 or more than 18 so Eligible procedure will execute otherwise if it is less than 18, Not Eligible procedure executes.

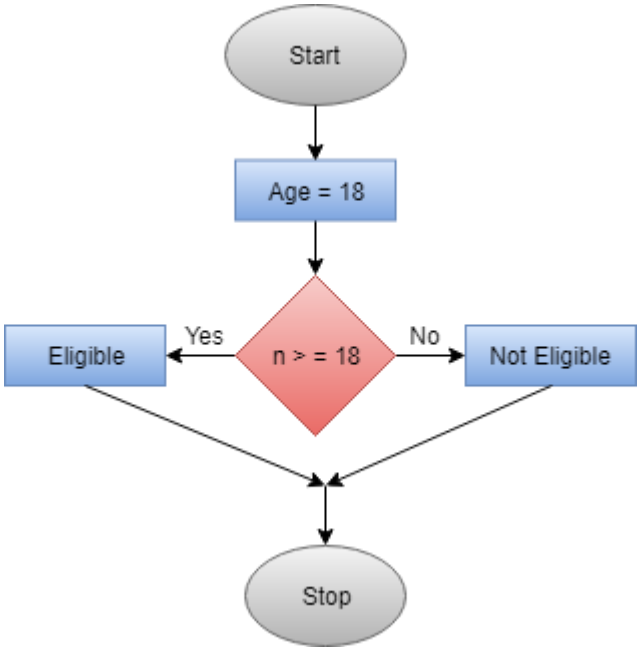
Junction node

Junction node in control flow graph is the point where at least three links meet.

Example

```
1. public class VoteEligiblityAge{
2.
3. public static void main(String []args){
4. int n=45;
5. if(n>=18)
6. {
7.     System.out.println("You are eligible for voting");
8.  }  else
9. {
10.          System.out.println("You are not eligible for voting");
11.        }
12.       }
13.       }
```

Diagram - control flow graph



In the control flow graph, start, age, eligible, not eligible and stop are the nodes, n>=18 is a decision node to decide which part (if or else) will execute as per the given value. Connectivity of the eligible node and not eligible node is there on the stop node.

Test cases are designed through the flow graph of the programs to determine the execution path is correct or not. All nodes, junction, edges, and decision are the essential parts to design test cases.

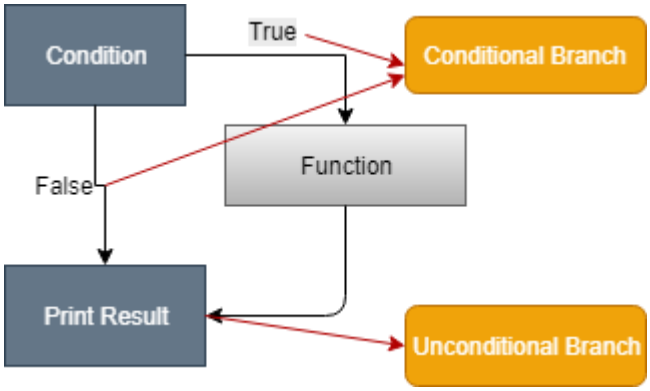
Branch Coverage Testing

Branch coverage technique is used to cover all branches of the control flow graph. It covers all the possible outcomes (true and false) of each condition of decision point at least once. Branch coverage technique is a whitebox testing technique that ensures that every branch of each decision point must be executed.

However, branch coverage technique and decision coverage technique are very similar, but there is a key difference between the two. Decision coverage technique covers all branches of each decision point whereas branch testing covers all branches of every decision point of the code.

In other words, branch coverage follows decision point and branch coverage edges. Many different metrics can be used to find branch coverage and decision coverage, but some of the most basic metrics

are: finding the percentage of program and paths of execution during the execution of the program.



How to calculate Branch coverage?

There are several methods to calculate Branch coverage, but pathfinding is the most common method.

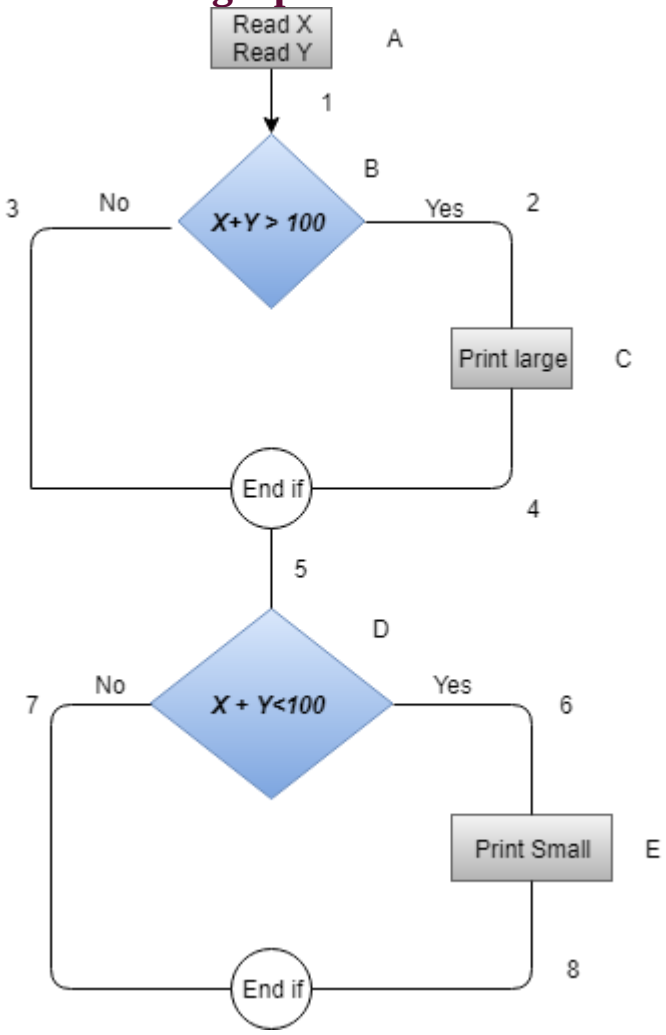
In this method, the number of paths of executed branches is used to calculate Branch coverage. Branch coverage technique can be used as the alternative of decision coverage. Somewhere, it is not defined as an individual technique, but it is distinct from decision coverage and essential to test all branches of the control flow graph.

Let's understand it with an example:

1. Read X
2. Read Y
3. IF X+Y > 100 THEN
4. Print "Large"
5. ENDIF
6. If X + Y<100 THEN
7. Print "Small"
8. ENDIF

This is the basic code structure where we took two variables X and Y and two conditions. If the first condition is true, then print "Large" and if it is false, then go to the next condition. If the second condition is true, then print "Small."

Control flow graph of code structure



In the above diagram, control flow graph of code is depicted. In the first case traversing through "Yes "decision, the path is **A1-B2-C4-D6-E8**, and the number of covered edges is 1, 2, 4, 5, 6 and 8 but edges 3 and 7 are not covered in this path. To cover these edges, we have to traverse through "No" decision. In the case of "No" decision the path is A1-B3-5-D7, and the number of covered edges is 3 and 7. So by traveling through these two paths, all branches have covered.

Path 1 - A1-B2-C4-D6-E8

Path 2 - A1-B3-5-D7

Branch Coverage (BC) = Number of paths

=2

Case	Covered Branches	Path	Branch coverage
Yes	1, 2, 4, 5, 6, 8	A1-B2-C4-D6-E8	2
No	3,7	A1-B3-5-D7	

Statement Coverage Testing

Statement coverage is one of the widely used software testing. It comes under white box testing.

Statement coverage technique is used to design white box test cases. This technique involves execution of all statements of the source code at least once. It is used to calculate the total number of executed statements in the source code out of total statements present in the source code.

Statement coverage derives scenario of test cases under the white box testing process which is based upon the structure of the code.

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

In white box testing, concentration of the tester is on the working of internal source code and flow chart or flow graph of the code.

Source Code Structure:

- Take input of two values like a=0 and b=1.
- Find the sum of these two values.
- If the sum is greater than 0, then print "This is the positive result."
- If the sum is less than 0, then print "This is the negative result."

1. input (**int** a, **int** b)
2. {
3. Function to print sum of these integer values (sum = a+b)
4. If (sum>0)
5. {
6. Print (This is positive result)
7. } **else**
8. {
9. Print (This is negative result)
10. }
11. }

So, this is the basic structure of the program, and that is the task it is going to do.

Now, let's see the two different scenarios and calculation of the percentage of Statement Coverage for given source code.

Scenario 1:
If a = 5, b = 4

1. print (**int** a, **int** b) {
2. **int** sum = a+b;
3. **if** (sum>0)
4. print ("This is a positive result")
5. **else**
6. print ("This is negative result")
7. }

1. Total number of statements = 7
2. Number of executed statements = 5

$$\text{Statement coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} * 100$$

1. Statement coverage = 5/7*100
2. = 500/7
3. = 71%

Decision Coverage Testing

Decision coverage technique comes under white box testing which gives decision coverage to Boolean values. This technique reports true and false outcomes of Boolean expressions. Whenever there is a possibility of two or more outcomes from the statements like **do while statement, if statement and case statement** (Control flow statements), it is considered as decision point because there are two outcomes either true or false.

Decision coverage covers all possible outcomes of each and every Boolean condition of the code by using control flow graph or chart.

Generally, a decision point has two decision values one is true, and another is false that's why most of the times the total number of outcomes is two. The percent of decision coverage can be found by dividing the number of exercised outcome with the total number of outcomes and multiplied by 100.

$$\textit{Decision Coverage} = \frac{\text{Number of Decision Outcomes Exercised}}{\text{Total Number of Decision Outcomes}} * 100$$

Let's understand it by an example.

Consider the code to apply on decision coverage technique:

```
1. Test (int a)
2. {
3. If(a>4)
4. a=a*3
5. Print (a)
6. }
```

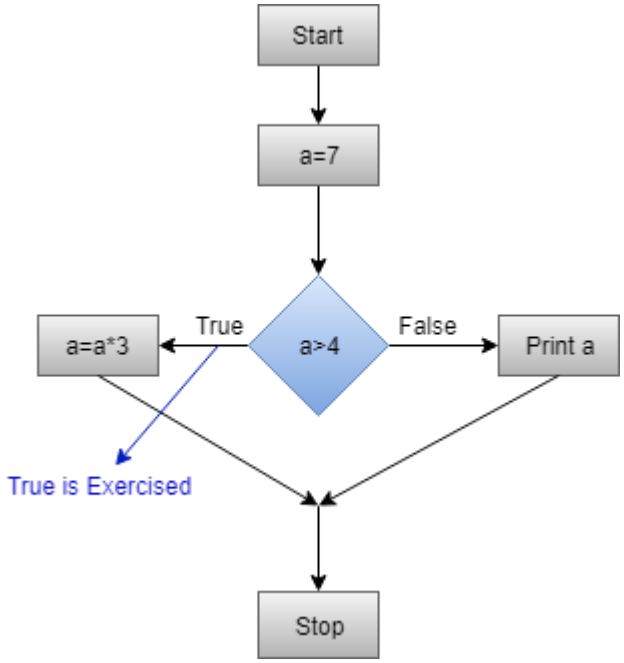
Scenario 1:
Value of a is 7 (a=7)

```
1. Test (int a=7)
2. { if (a>4)
```

3. a=a*3
4. print (a)
5. }

The code highlighted in yellow is executed code. The outcome of this code is "True" if condition (a>4) is checked.

Control flow graph when the value of a is 7.



Calculation of Decision Coverage percent:

$$\textit{Decision Coverage} = \frac{\text{Number of Decision Outcomes Exercised}}{\text{Total Number of Decision Outcomes}} * 100$$

1. Decision Coverage = ½*100 (Only "True" is exercised)
2. =100/2
3. = 50
4. Decision Coverage is 50%