- **What is the name of Javascript Engines for ‘Chrome’, ‘Firefox’, ‘Edge’?**
    - Chrome: V8
    - Firefox: SpiderMonkey
    - Edge: Chakra (Legacy) / V8 (New Edge uses the same engine as Chrome, i.e., V8)

- **What is the purpose of a ‘full-codegen’ baseline compiler?**
    - The 'full-codegen' baseline compiler is designed to quickly generate **machine code from JavaScript source code**. It prioritizes speed of compilation over optimizing the generated code. It helps execute code faster but sacrifices potential performance improvements that could be achieved through more advanced optimization techniques.

- **What was the main drawback of the first gen V8 model?**
    - The main drawback of the first-generation V8 model was that it lacked proper support for asynchronous operations. JavaScript, being single-threaded, often requires asynchronous execution patterns (like callbacks, promises, async/await) to prevent blocking behavior. The first-gen V8 model struggled to efficiently handle these patterns, leading to performance issues and complex code.

- **What are the main components of Javascript at Runtime?**
- At runtime, the main components of JavaScript include:
    1) Heap: Memory where objects are allocated.
    2) Call Stack: Keeps track of the execution context of functions.
    3) Event Loop: Manages asynchronous operations and callbacks.
    4) Callback Queue: Holds callbacks and events to be processed.
    5) Web APIs: Provides browser-specific functionality.
    6) Microtask Queue: Holds microtasks, which have higher priority than tasks from the callback queue.
    -
- **What is the event loop in JavaScript engines?**
    - The event loop is a crucial part of JavaScript's concurrency model. It continuously checks the callback queue for tasks to be executed. If the call stack is empty, it pops tasks from the queue and pushes them onto the call stack for execution. This allows asynchronous operations to be executed in a non-blocking manner.

- **Define the call-stack for the following code snippet.**
  const second = () => {
  console.log(‘Hello there!’);
  }
  const first = () => {
  console.log(‘Hi there!’);
  second();
  console.log(‘The End’);
  }
  first();
    - first() is pushed onto the call stack.
      Inside first(), console.log('Hi there!') is executed.
      second() is pushed onto the call stack.
      Inside second(), console.log('Hello there!') is executed.
      second() is popped from the call stack.
      Inside first(), console.log('The End') is executed.
      first() is popped from the call stack.

- **What is the output for the following code snippet?**
  console.log(“Hello.”);
  setTimeout(function() {
  console.log(“Goodbye!”);
  }, 3000);
  console.log(“Hello again!”)
    - Hello.
      Hello again!
      Goodbye! (after a 3-second delay)

- **Explain how the following code executes with javascript engine.**

```
function printHello() {
console.log('Hello from baz');
}
function baz() {
 setTimeout(printHello, 3000);
}
function bar() {
  baz();
}
function foo() {
bar();
}
foo();
```

  - The code defines four functions: printHello, baz, bar, and foo. The foo function calls bar, which calls baz, and finally, baz schedules printHello to be executed after a 3-second delay. Therefore, after 3 seconds, "Hello from baz" will be logged to the console.

- **Output of Following JS :**

```
function a() {
        setTimeout( function() {
      console.log( 'result of a()' );
  }, 1000 );
}

function b() {
        setTimeout( function() {
      console.log( 'result of b()' );
  }, 500 );
}

function c() {
        setTimeout( function() {
      console.log( 'result of c()' );
```

```
    }, 1200 );
}

// call in sequence
a();
console.log('a() is done!');

b();
console.log('b() is done!');

c();
console.log('c() is done!');
```

Answer :

a() is done!
b() is done!
c() is done!
result of b()
result of a()
result of c()

- **Output of following Question :**

```
function a( callback ) {
        setTimeout( () => {
    console.log( 'result of a()' );
    callback();
  }, 1000 ); // 1 second delay
}

function b( callback ) {
        setTimeout( () => {
    console.log( 'result of b()' );
    callback();
  }, 500 ); // 0.5 second delay
}
```

```
function c( callback ) {
        setTimeout( () => {
      console.log( 'result of c()' );
      callback();
   }, 1200 ); // 1.1 second delay
}

// call in sequence
a( () => console.log('a() is done!') );
b( () => console.log('b() is done!') );
c( () => console.log('c() is done!') );
```

Answer :

result of a()
a() is done!
result of b()
b() is done!
result of c()
c() is done!

- **Output of following JS**

```
function a( callback ) {
        setTimeout( () => {
      console.log( 'result of a()' );
      callback();
   }, 1000 ); // 1 second delay
}

function b( callback ) {
        setTimeout( () => {
      console.log( 'result of b()' );
      callback();
   }, 500 ); // 0.5 second delay
}

function c( callback ) {
```

```
        setTimeout( () => {
        console.log( 'result of c()' );
        callback();
    }, 1200 ); // 1.1 second delay
}

// call in sequence
a( () => {
    console.log('a() is done!');
    b (()=> {
        console.log('b() is done!');
        c (()=>{
            console.log('c() is done!')
        })
    })
} );
```

Answer :

result of a()
a() is done!
result of b()
b() is done!
result of c()
c() is done!

Question :

```
const promiseA = new Promise( ( resolve, reject ) => {
    setTimeout( () => {
        // resolve('a() was successful');
        reject( 'something bad happened a()' );
    }, 1000 ); // reject after 1 second
} );

promiseA
.then( ( result ) => {
```

```
    console.log( 'promiseA success:', result );
} )
.catch( ( error ) => {
    console.log( 'promiseA error:', error );
} )
.finally( () => {
    console.log('a() is done!');
} );
```

Answer :

promiseA error: something bad happened a()
a() is done!

Explain : The promiseA is created with a setTimeout function that simulates an asynchronous operation. After a 1-second delay, the reject function is called with the message 'something bad happened a()', indicating that the operation has failed with an error.
The promiseA is then used with .then() to handle the successful resolution and .catch() to handle the error. In this case, the error path is taken because reject was called.
The catch block is executed, and it logs 'promiseA error: something bad happened a()'.
The .finally() block is always executed, regardless of whether the promise is fulfilled or rejected. In this case, it logs 'a() is done!'.

Question :

```
// setTimeout with `0` delay
setTimeout( () => console.log( 'setTimeout callback' ), 0 );

// immediately resolved promise
const promiseA = new Promise( ( resolve ) => resolve() );

// normal flow
console.log( 'I am sync job.' );

// promise listener
promiseA.then( () => {
```

```
    console.log( 'promiseA success:' );
} );

// normal flow
console.log( 'I am good sync job.' );
console.log( 'I am awesome sync job too.' );
```

Answer :

I am sync job.
I am good sync job.
I am awesome sync job too.
setTimeout callback
promiseA success:

Explain :

The use of setTimeout with 0 delay doesn't mean the callback will be executed instantly. It will be executed as soon as the main thread is free and other synchronous code is completed. JavaScript's event loop ensures that all synchronous code is executed first before handling the callbacks from the callback queue, including the resolved promises.