JavaScript Quiz

- Event Loop - 4 questions
- Scope & Closures - 6 questions
- this Keyword - 5 questions
- Classes and Prototypes - 5 questions
- Generators & Iterators - 4 questions
- Garbage Collection - 6 questions
- Modules - 3 questions
- Miscellaneous - 17 questions

Event Loop

```
1  console.log(1);
2
3  function myFunction() {
4    console.log(2);
5  }
6
7  myFunction();
```

console

Call Stack

Web API

Event Loop

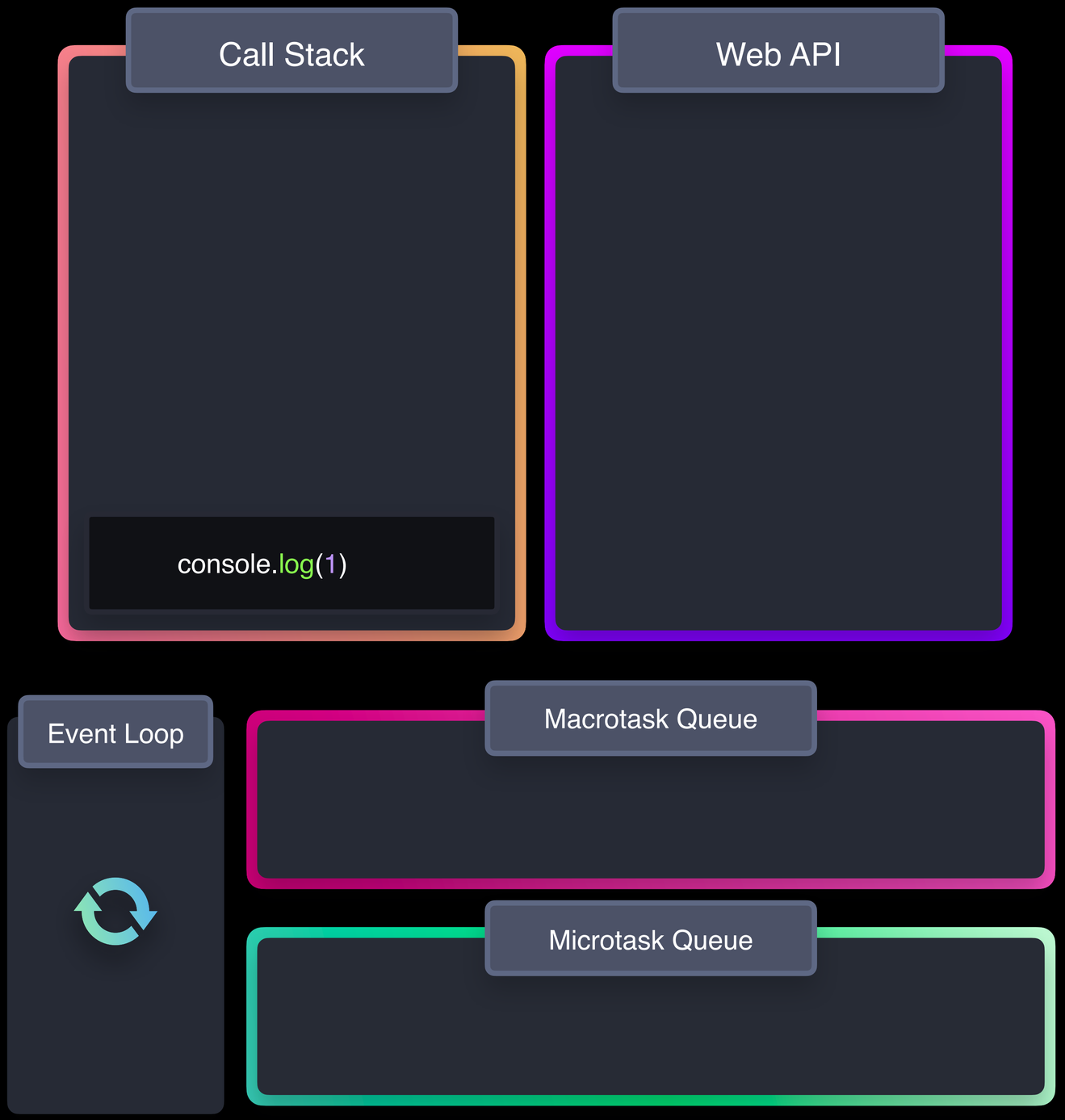
Macrotask Queue

Microtask Queue

```
1  console.log(1);
2
3  function myFunction() {
4    console.log(2);
5  }
6
7  myFunction();
```

console

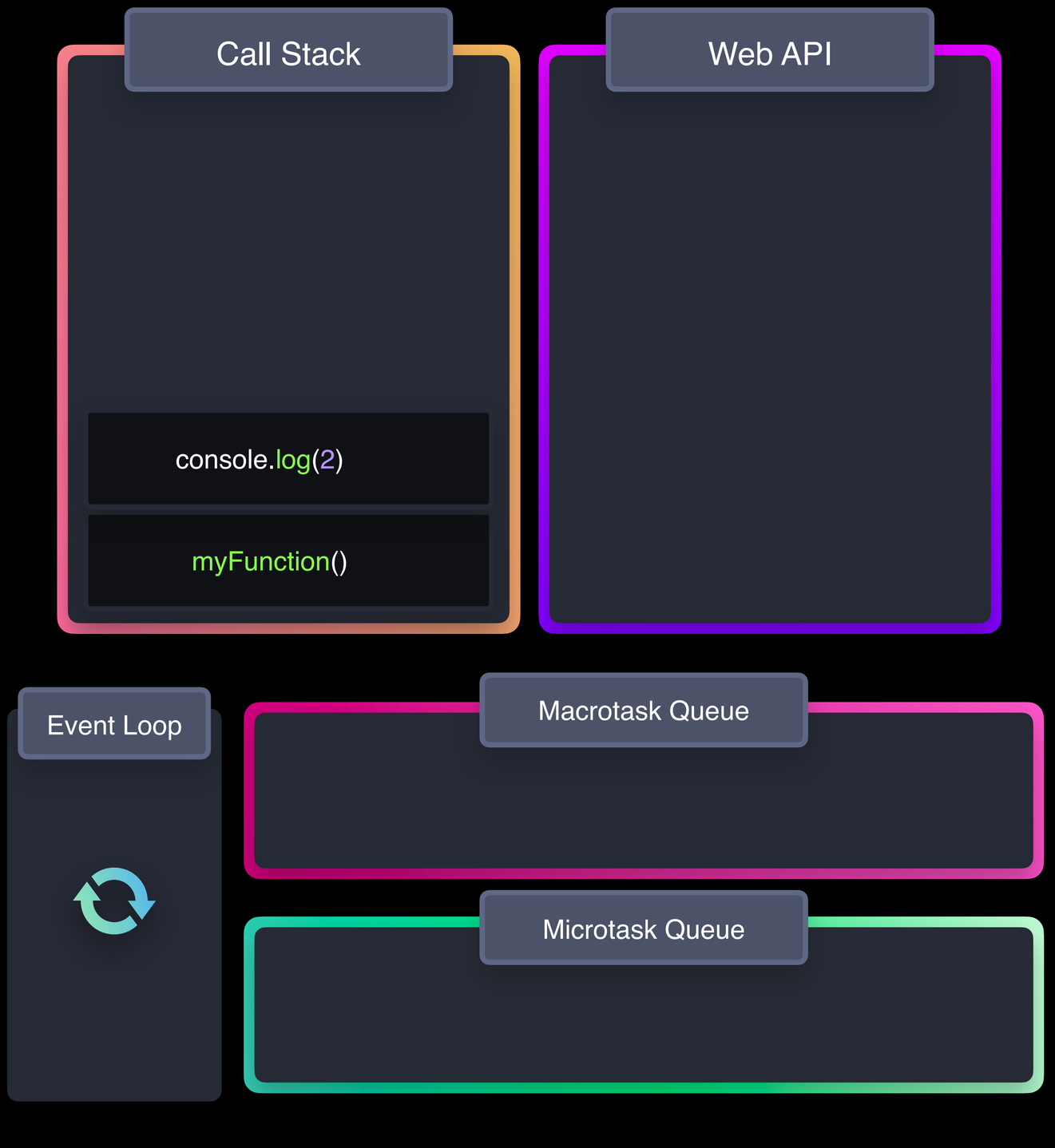
1



```
1  console.log(1);
2
3  function myFunction() {
4    console.log(2);
5  }
6
7  myFunction();
```

console

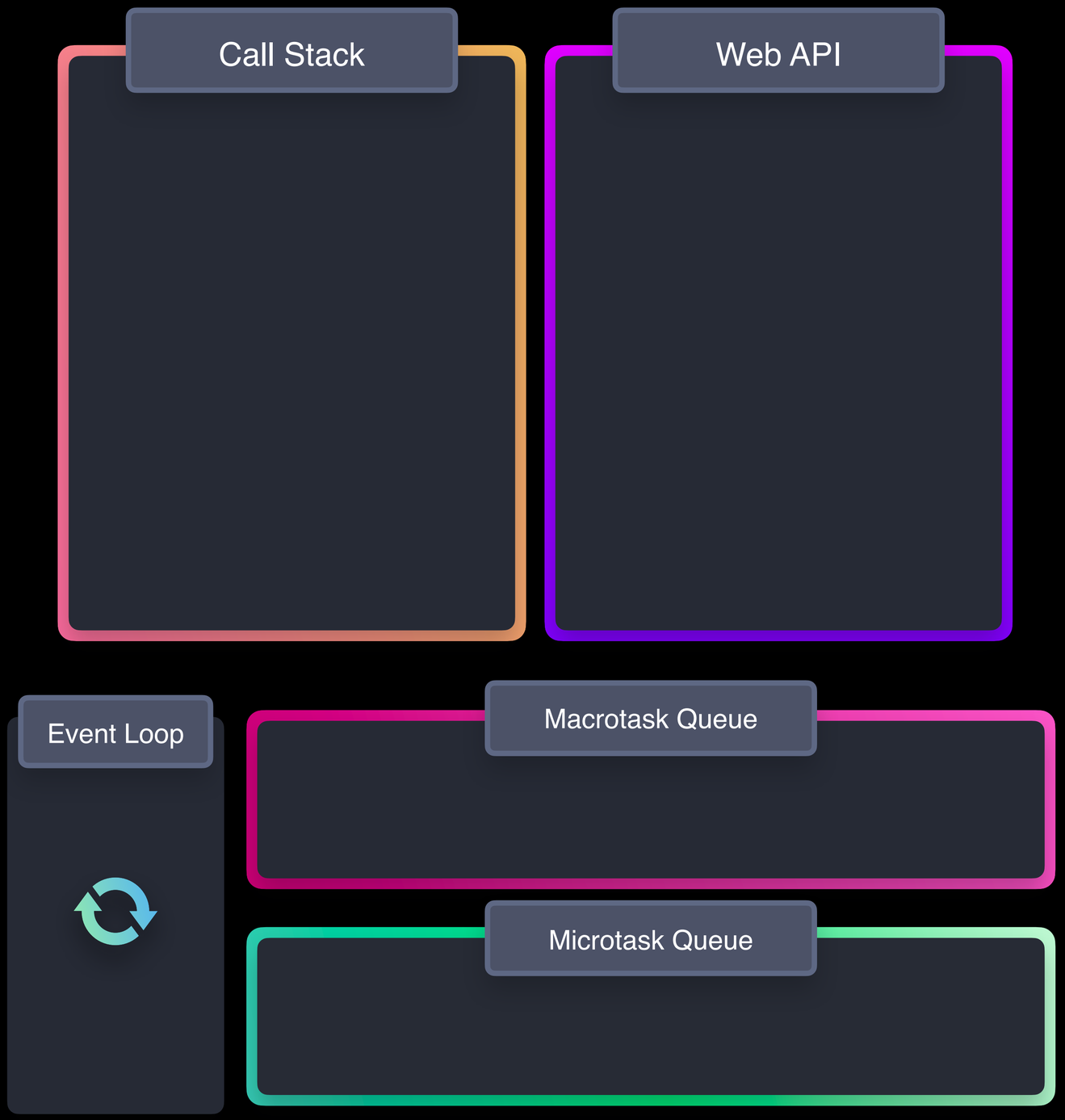
```
1
2
```



```
1  console.log(1);
2
3  function myFunction() {
4    console.log(2);
5  }
6
7  myFunction();
```

console

```
1
2
```



A WebAPI to **schedule callback to macrotask queue**

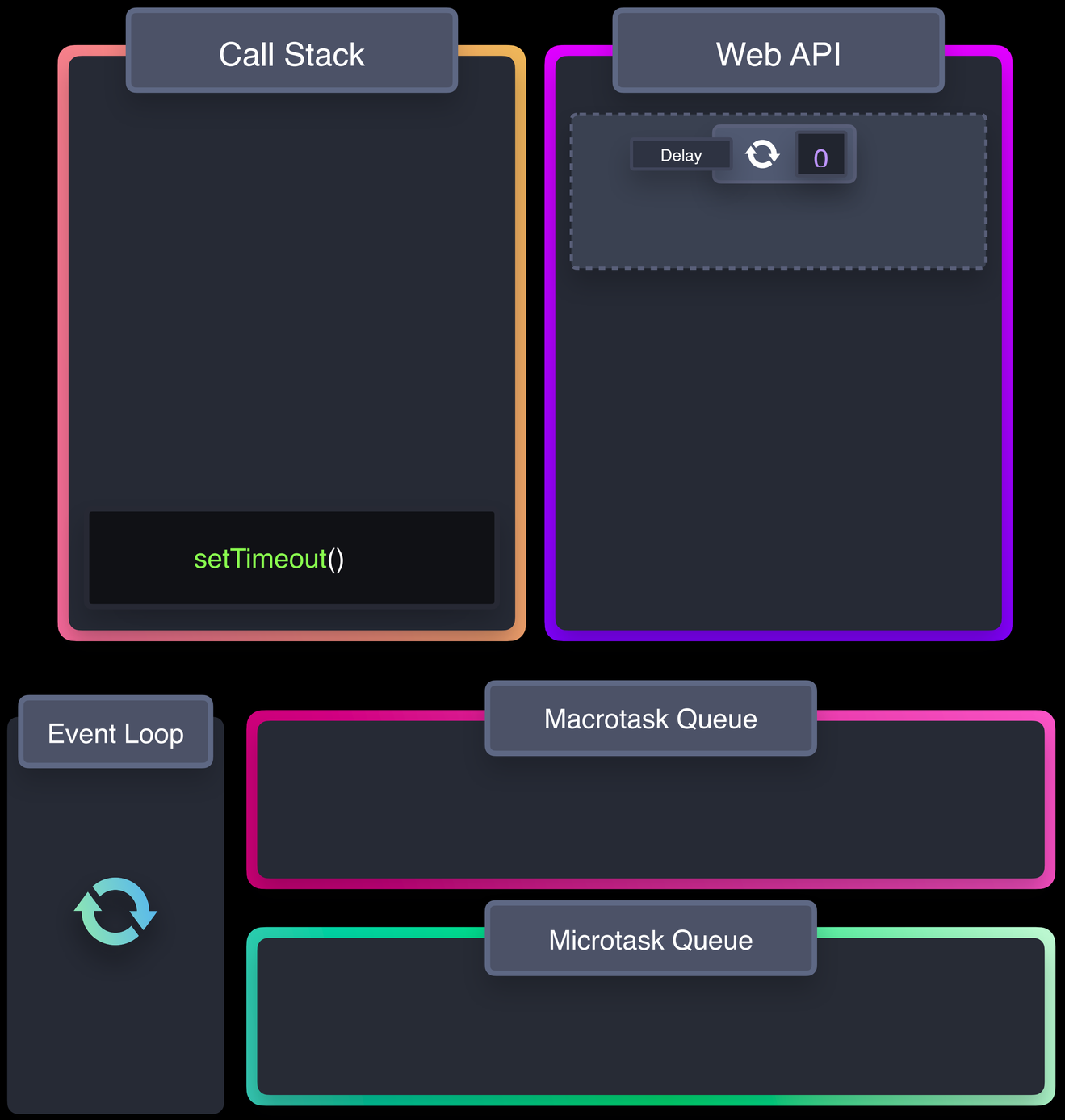
Callback to be pushed to the macrotask queue

Delay

```
setTimeout(() => console.log(1), 100);
```

```
1  setTimeout(() => console.log(1), 0);
```

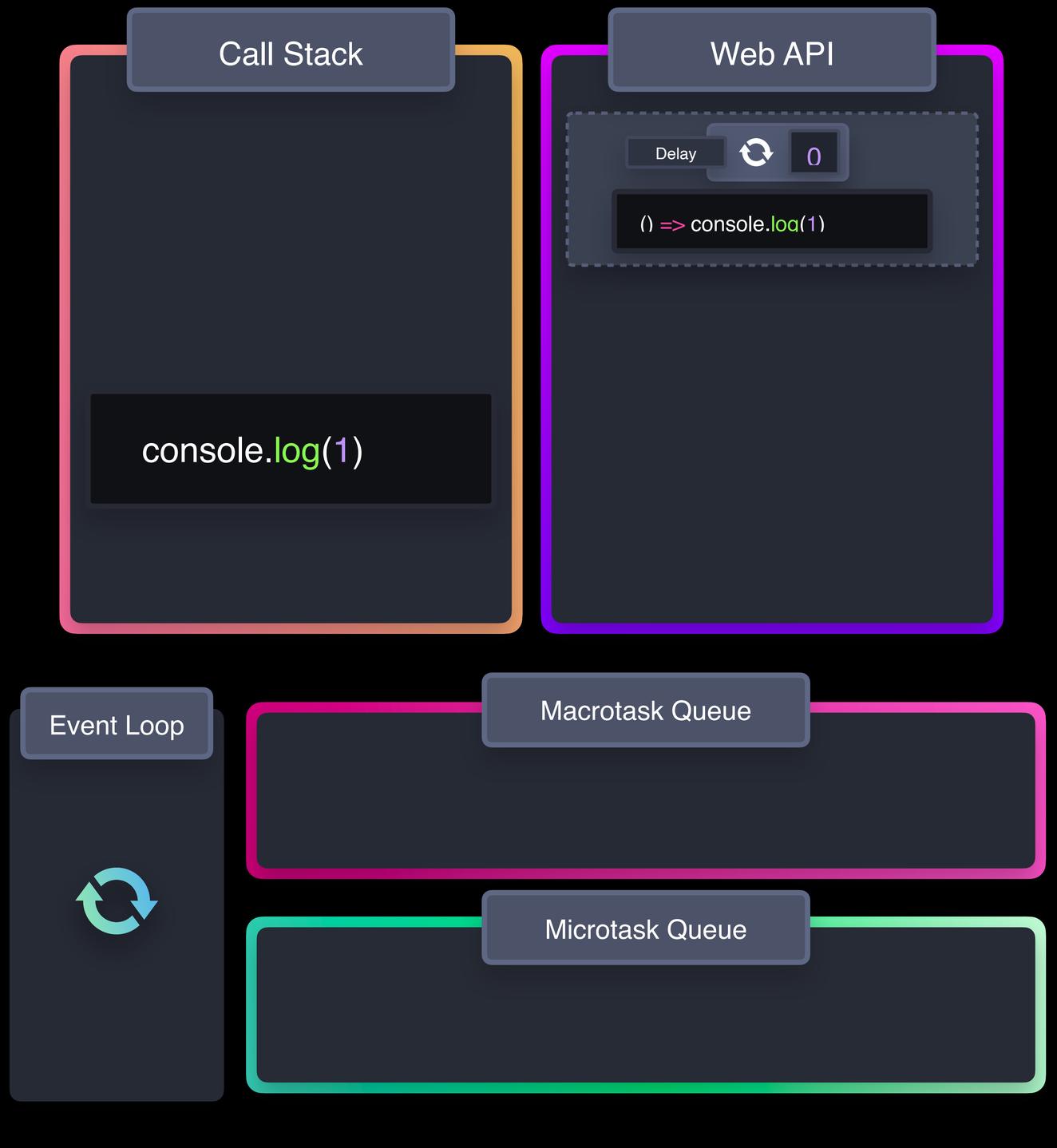
console



```
1  setTimeout(() => console.log(1), 0);
```

console

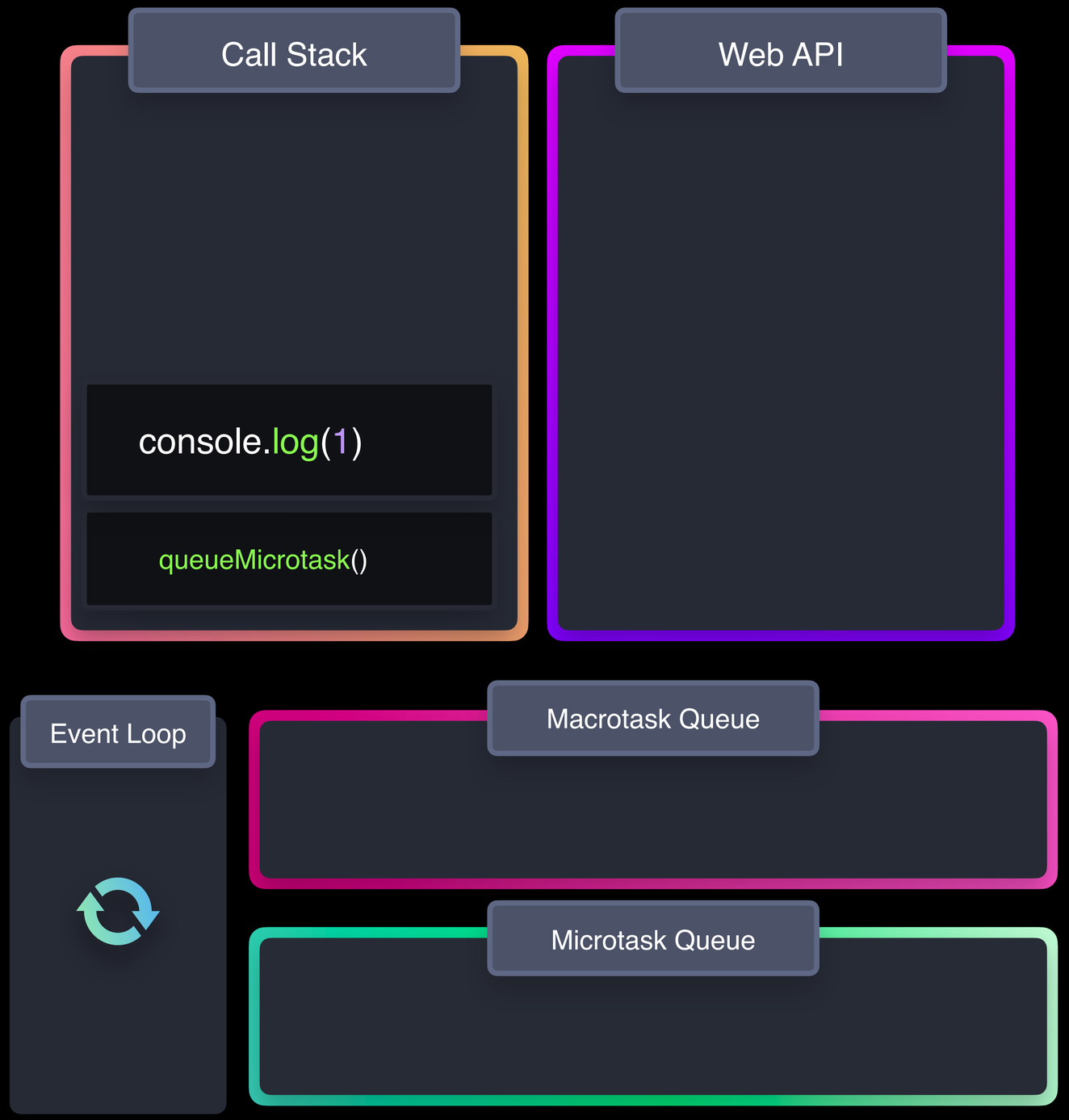
1

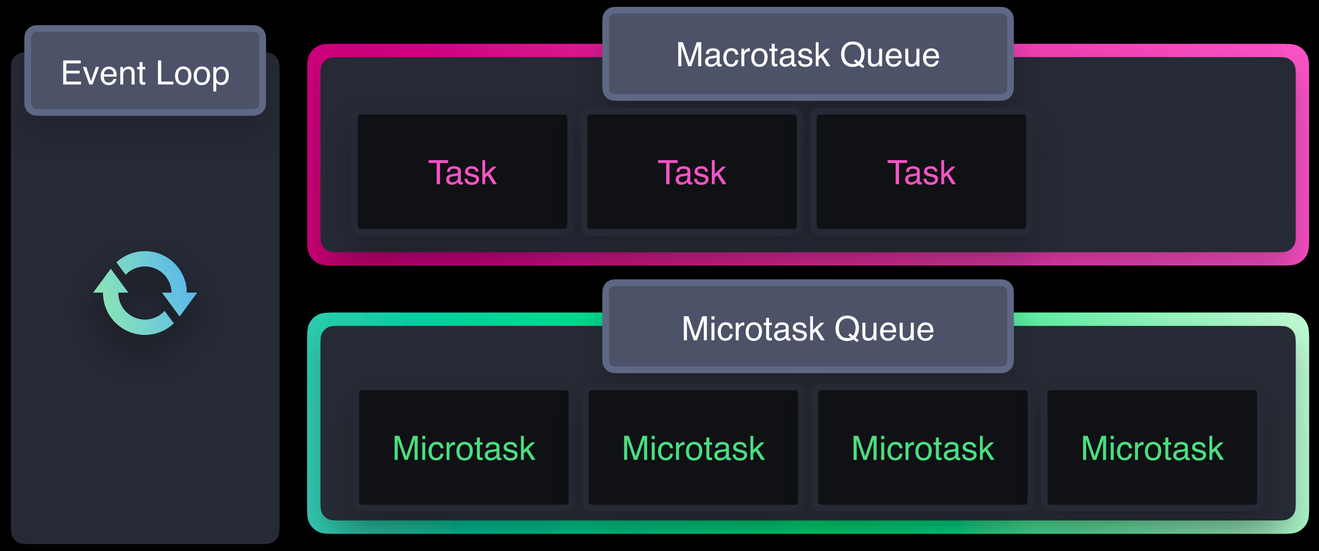


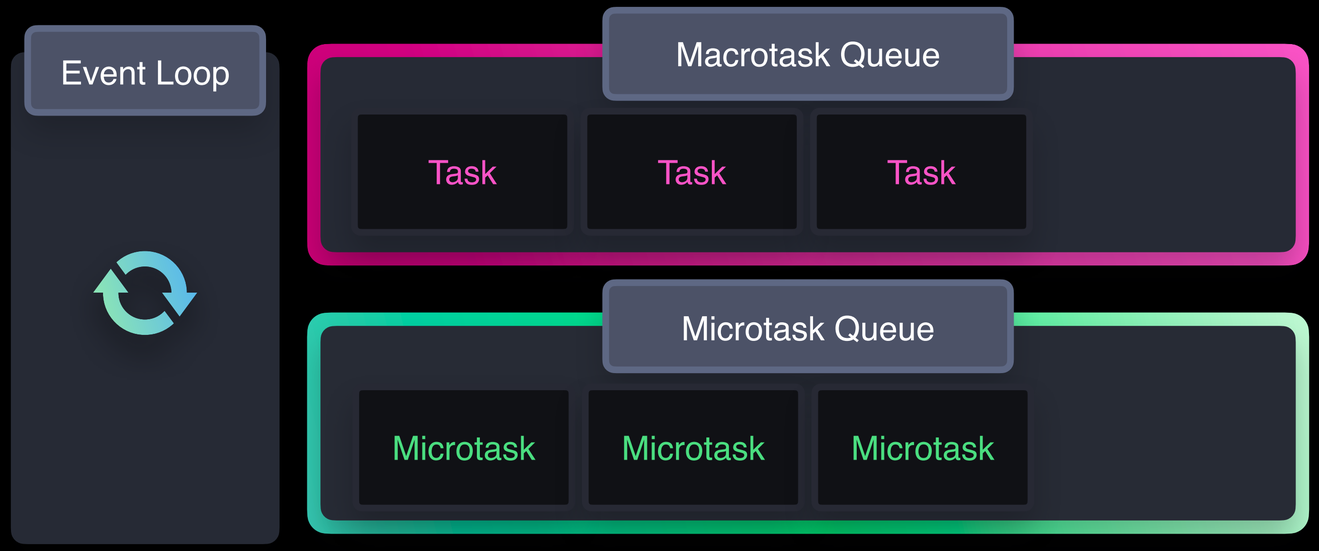
```
1  queueMicrotask(() => console.log(1));
```

console

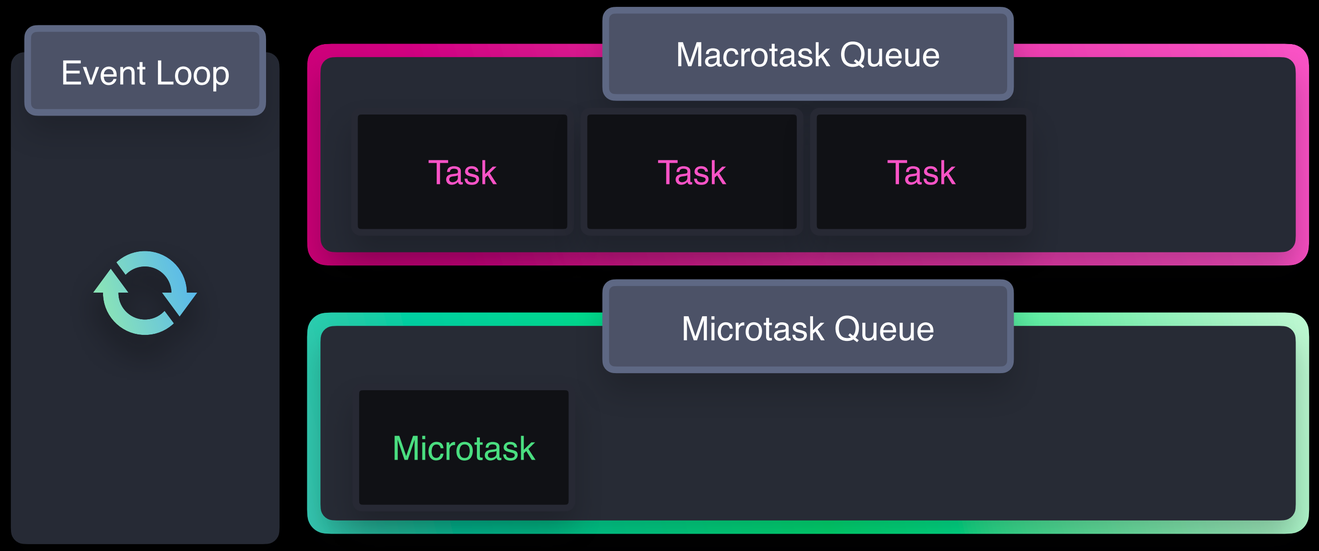
1

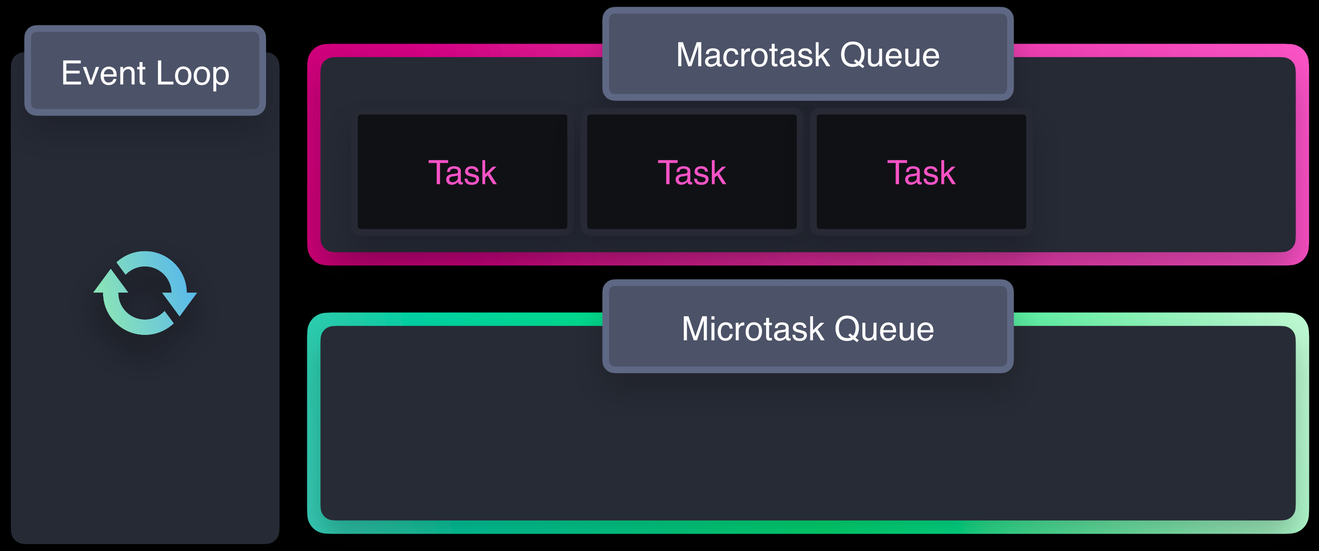


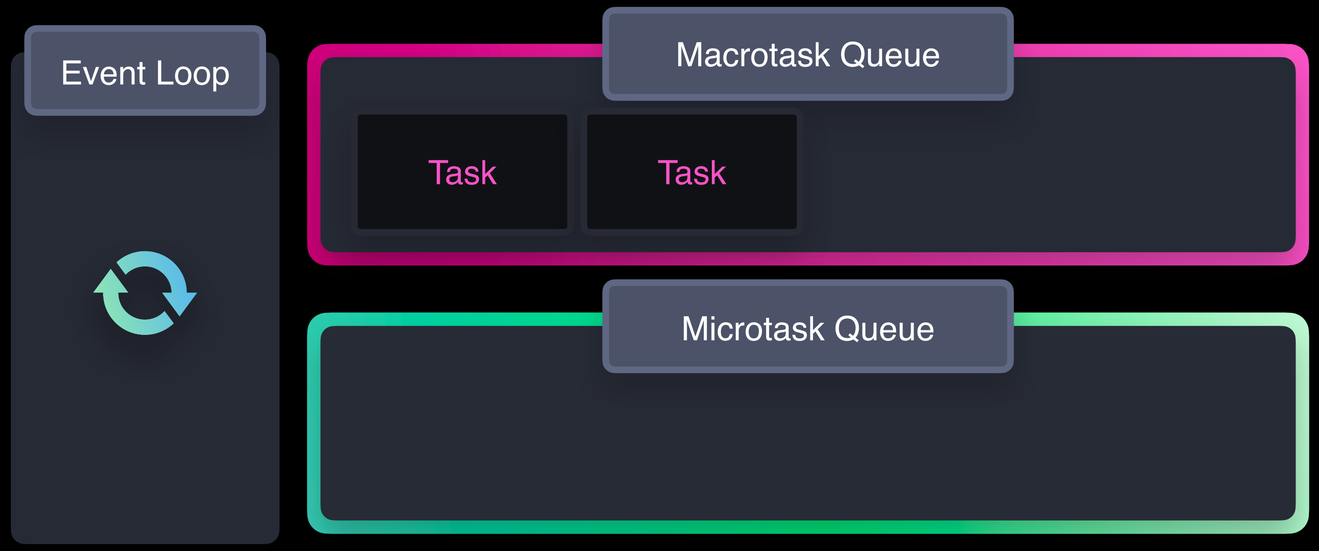

Event Loop
Macrotask Queue
Task
Task
Task
Microtask Queue
Microtask
Microtask
Microtask
Microtask

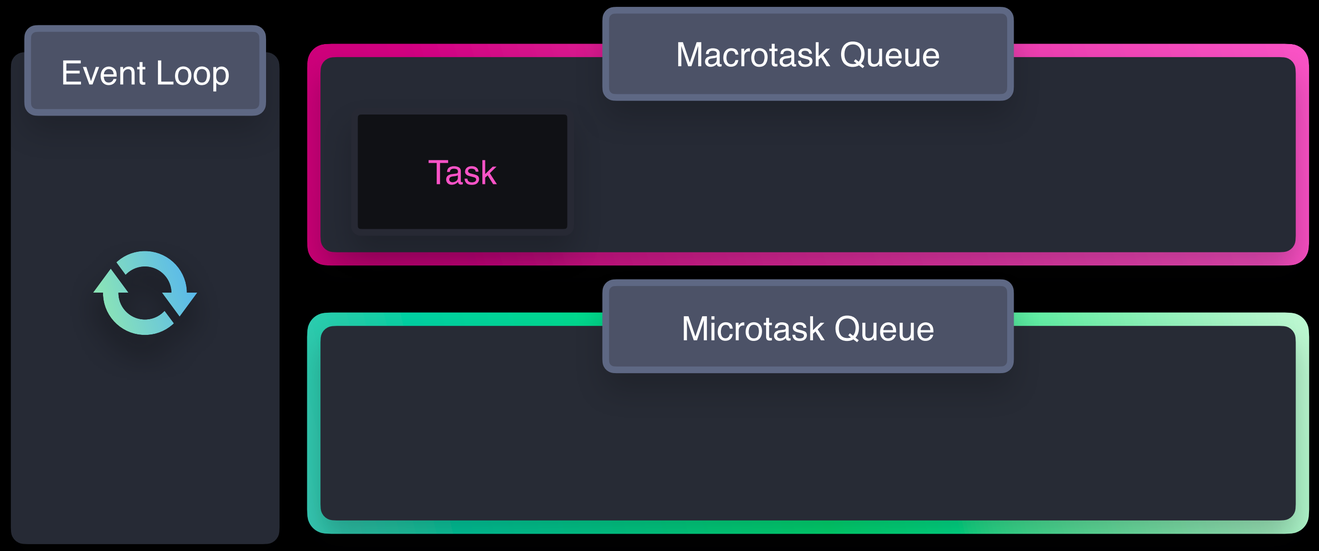

Event Loop
Macrotask Queue
Task
Task
Task
Microtask Queue
Microtask
Microtask
Microtask

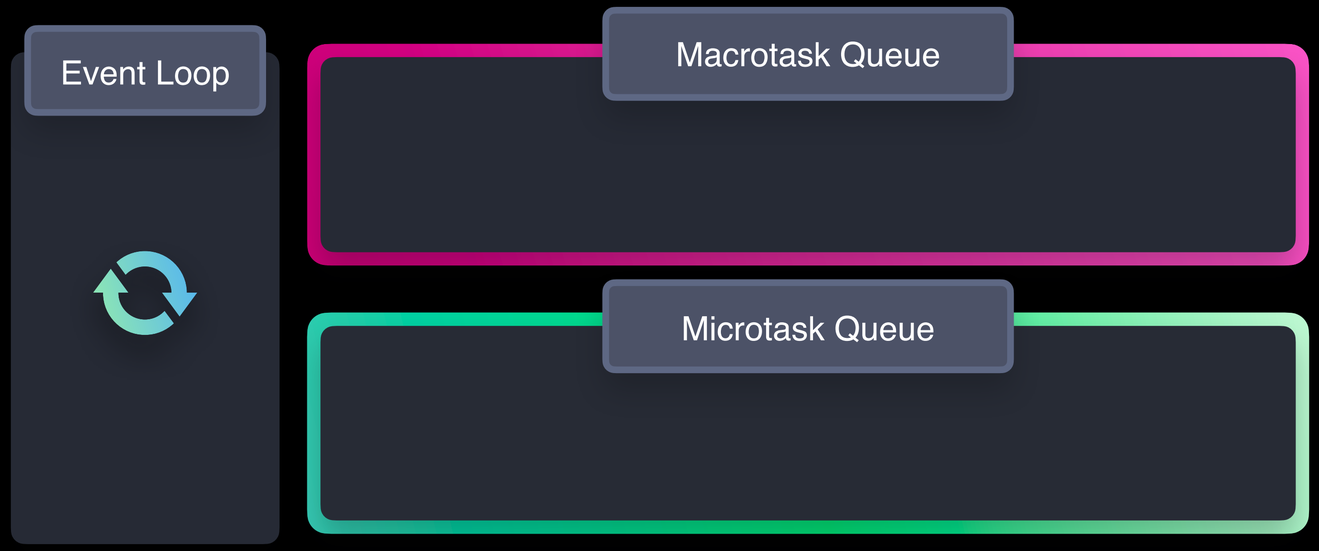
Event Loop
Macrotask Queue
Task
Task
Task
Microtask Queue
Microtask
Microtask


Event Loop
Macrotask Queue
Task
Task
Task
Microtask Queue
Microtask


Event Loop
Macrotask Queue
Task
Task
Task
Microtask Queue


Event Loop
Macrotask Queue
Task
Task
Microtask Queue


Event Loop
Macrotask Queue
Task
Microtask Queue


Event Loop
Macrotask Queue
Microtask Queue

```
async function asyncFunc() {
    console.log(1);
    await console.log(2);
    console.log(3);
}
```

```
.then(    () => console.log(1)    );
.catch(    () => console.log(2)    );
.finally(    () => console.log(3)    );
```

```
queueMicrotask(    () => console.log(1)    );
```

```
new MutationObserver(    () => console.log(1)    );
```

```
process.nextTick(    () => console.log(1)    );
```

Question 1

Put the logs in the correct order

```
Promise.resolve()
  .then(() => console.log(1));

queueMicrotask(() => console.log(2));

setTimeout(() => console.log(3), 0);

console.log(4);

new Promise(() => console.log(5));

(async () => console.log(6))()
```

1

2

3

4

5

6

Question 1

Put the logs in the correct order

```
1   Promise.resolve()
2     .then(() => console.log(1));
3
4   queueMicrotask(() => console.log(2));
5
6   setTimeout(() => console.log(3), 0);
7
8   console.log(4);
9
10  new Promise(() => console.log(5));
11
12  (async () => console.log(6))()
```

4

5

6

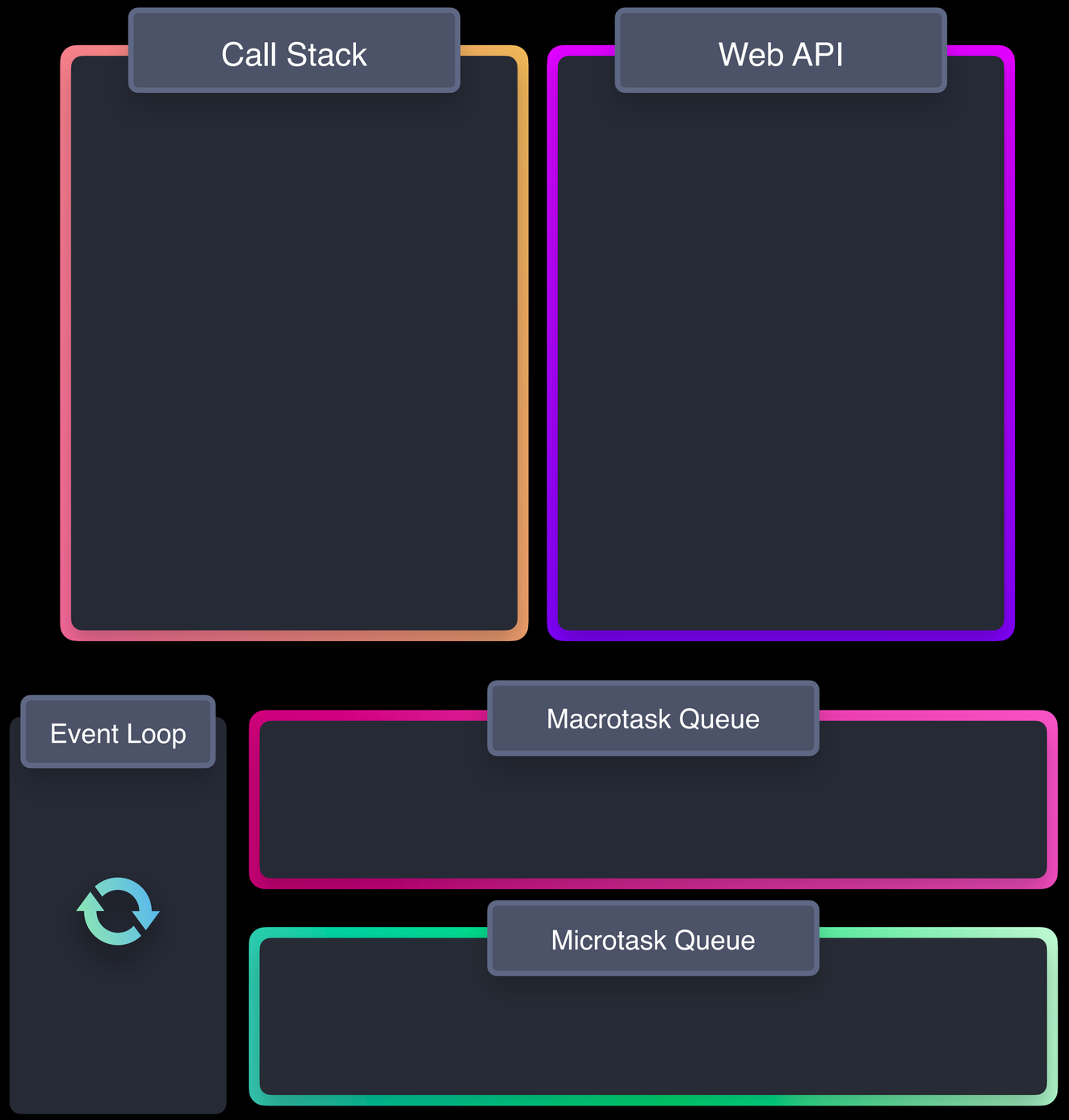
1

2

3

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console



```
Promise.resolve()
  .then(() => console.log(1));

queueMicrotask(() => console.log(2));

setTimeout(() => console.log(3), 0);

console.log(4);

new Promise(() => console.log(5));

(async () => console.log(6))()
```

console

Call Stack

Web API

Event Loop

Macrotask Queue

Microtask Queue

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console

Call Stack

Promise.resolve()

Web API

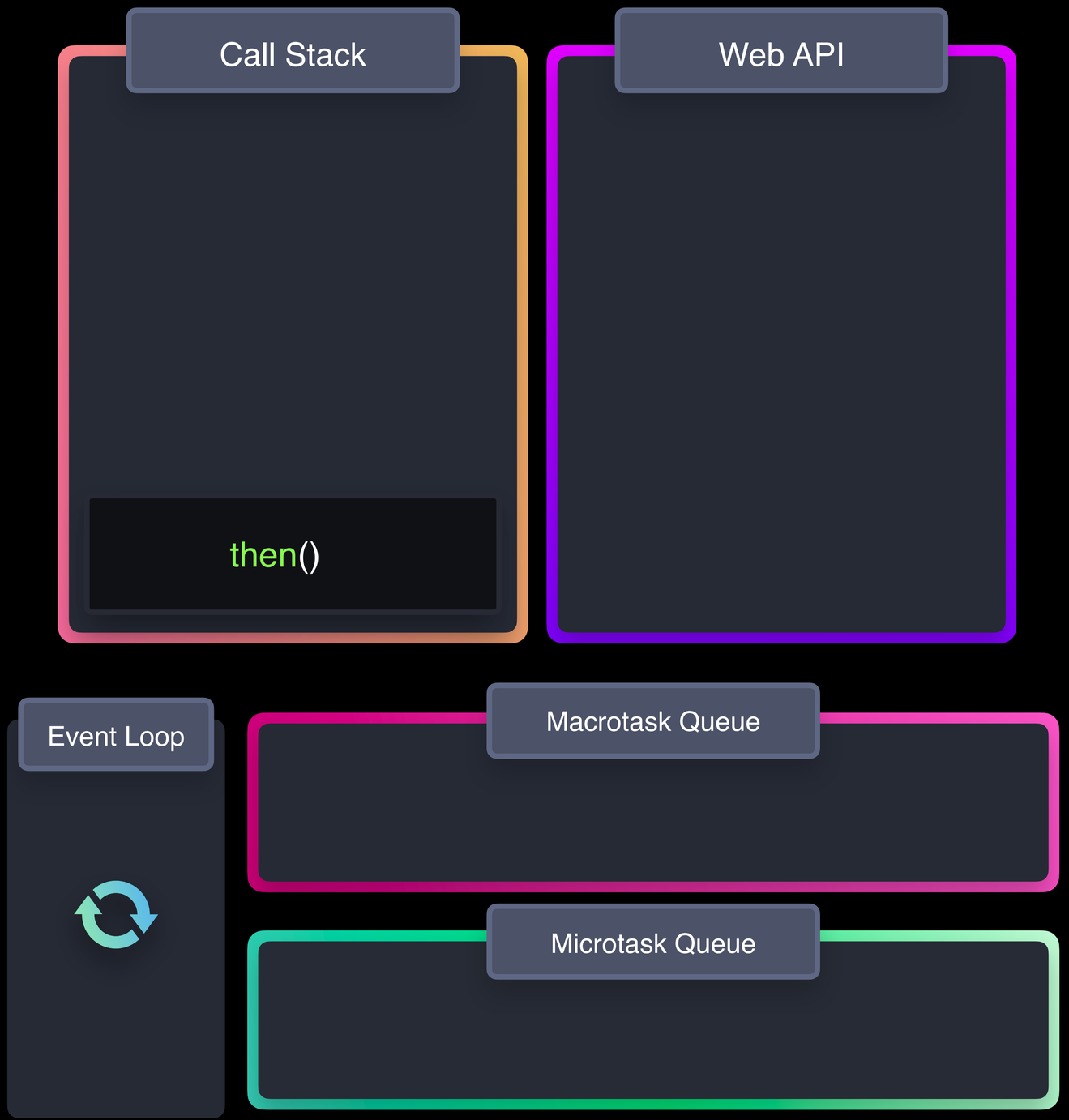
Event Loop

Macrotask Queue

Microtask Queue

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console



```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console

Call Stack

queueMicrotask()

Web API

Event Loop

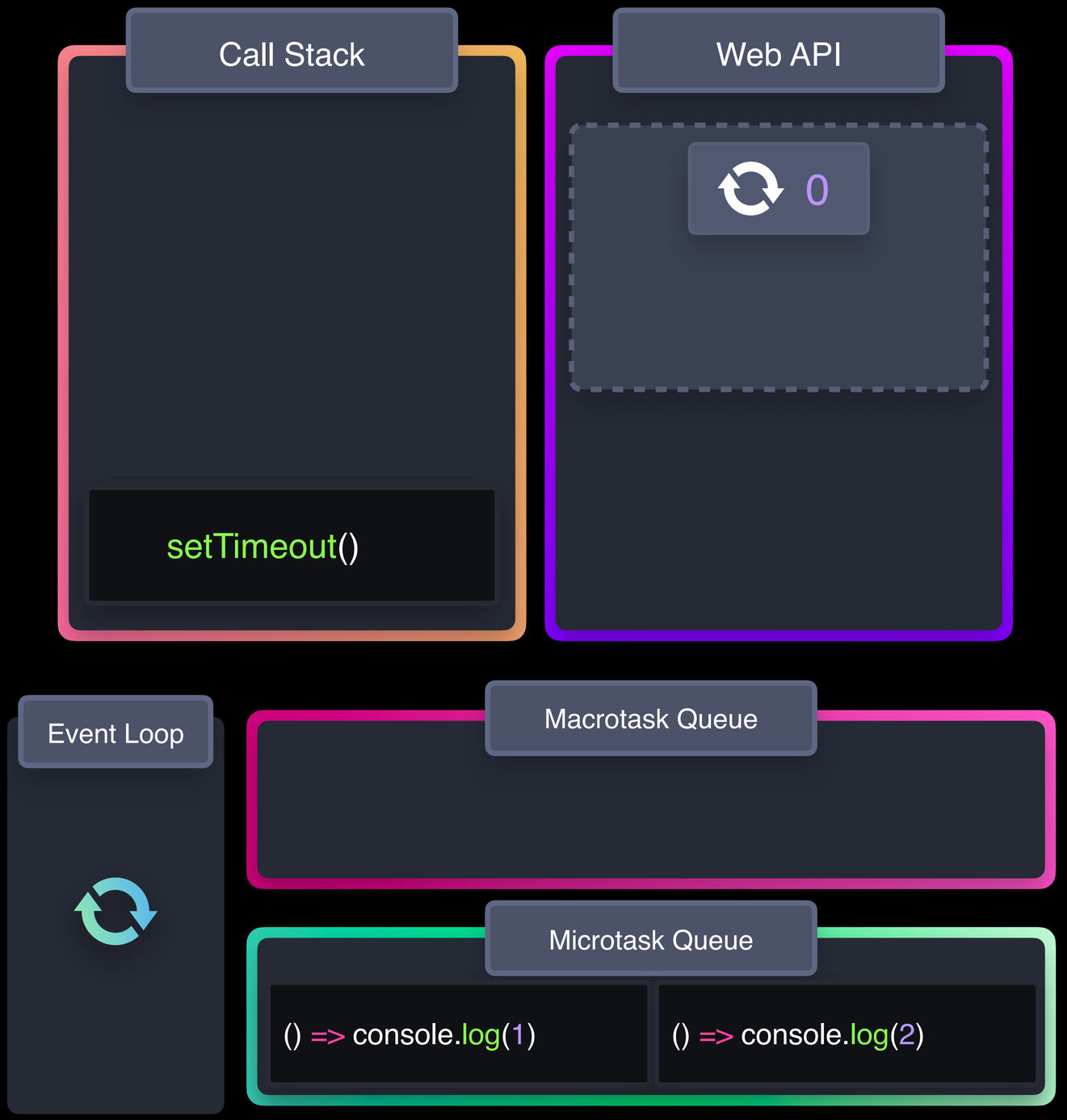
Macrotask Queue

Microtask Queue

() => console.log(1)

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console



```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console

4

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console

4
5

Call Stack

console.log(5)

new *Promise*(
() => console.log(5)
)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

() => console.log(1)

() => console.log(2)

```
1   Promise.resolve()
2     .then(() => console.log(1));
3
4   queueMicrotask(() => console.log(2));
5
6   setTimeout(() => console.log(3), 0);
7
8   console.log(4);
9
10  new Promise(() => console.log(5));
11
12  (async () => console.log(6))()
```

console

4
5
6

```
Promise.resolve()
  .then(() => console.log(1));

queueMicrotask(() => console.log(2));

setTimeout(() => console.log(3), 0);

console.log(4);

new Promise(() => console.log(5));

(async () => console.log(6))()
```

console

4
5
6

Call Stack

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

() => console.log(1)

() => console.log(2)

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console

4
5
6
1

Call Stack

console.log(1)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

() => console.log(1)

() => console.log(2)

```
1  Promise.resolve()
2    .then(() => console.log(1));
3
4  queueMicrotask(() => console.log(2));
5
6  setTimeout(() => console.log(3), 0);
7
8  console.log(4);
9
10 new Promise(() => console.log(5));
11
12 (async () => console.log(6))()
```

console

4
5
6
1
2

Call Stack

console.log(2)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

() => console.log(2)

```
Promise.resolve()
  .then(() => console.log(1));

queueMicrotask(() => console.log(2));

setTimeout(() => console.log(3), 0);

console.log(4);

new Promise(() => console.log(5));

(async () => console.log(6))()
```

console

4
5
6
1
2
3

Call Stack

console.log(3)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

```
1   Promise.resolve()
2     .then(() => console.log(1));
3
4   queueMicrotask(() => console.log(2));
5
6   setTimeout(() => console.log(3), 0);
7
8   console.log(4);
9
10  new Promise(() => console.log(5));
11
12  (async () => console.log(6))()
```

console

4
5
6
1
2
3

Call Stack

Web API

Event Loop

Macrotask Queue

Microtask Queue

Question 2

Which of the following are not considered microtasks in JavaScript's event loop?

A	Script loading	E	UI rendering
B	setTimeout callback	F	then callback
C	mousemove event	G	new Promise executor function
D	requestAnimationFrame callback	H	fetch calls

Question 2

Which of the following are not considered microtasks in JavaScript's event loop?

- **A** Script loading
- **B** setTimeout callback
- **C** mousemove event
- **D** requestAnimationFrame callback
- **E** UI rendering
- **F** then callback
- **G** new Promise executor function
- **H** fetch calls

```
async function asyncFunc() {
    console.log(1);
    await console.log(2);
    console.log(3);
}
```

```
.then(    () => console.log(1)    );
.catch(    () => console.log(2)    );
.finally(    () => console.log(3)    );
```

```
queueMicrotask(    () => console.log(1)    );
```

```
new MutationObserver(    () => console.log(1)    );
```

```
process.nextTick(    () => console.log(1)    );
```

Question 3

Put the logs in the correct order

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

1	5
2	6
3	7
4	8

Question 3

Put the logs in the correct order

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

4

5

1

2

8

7

6

3

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

Call Stack

Web API

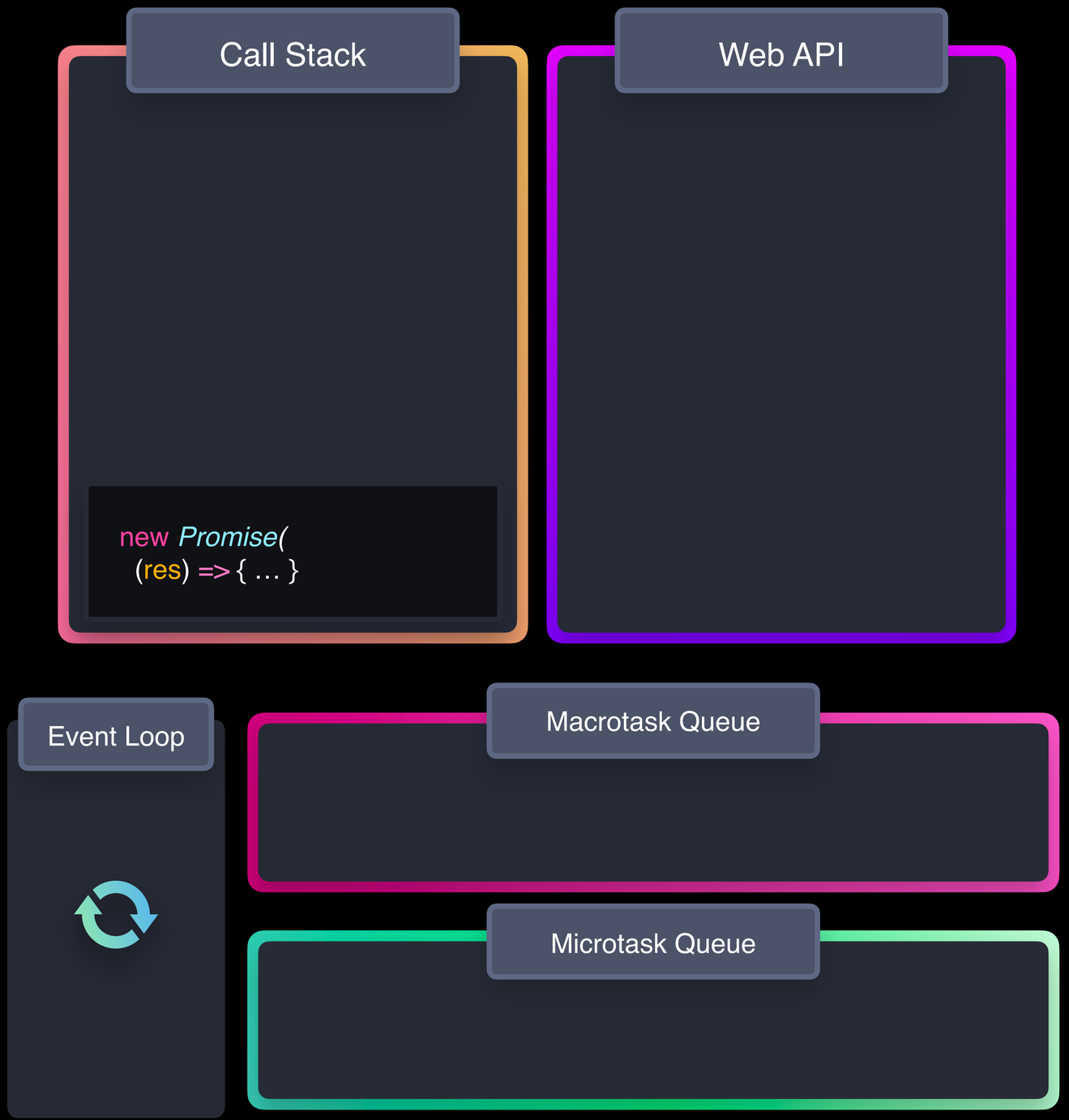
Event Loop

Macrotask Queue

Microtask Queue

```javascript
1  async function asyncFunction() {
2    console.log(1);
3    new Promise(() => console.log(2));
4    await new Promise((res) => {
5      setTimeout(() => console.log(3), 0);
6      res();
7    });
8  };
9
10 new Promise((res) => {
11   console.log(4);
12     (async () => {
13       console.log(5);
14       await asyncFunction();
15       console.log(6);
16     })();
17   res();
18 }).then(() => console.log(7));
19
20 console.log(8);
```

console



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4

Call Stack

console.log(4);

new *Promise(*
(res) => { … }

Web API

Event Loop

Macrotask Queue

Microtask Queue

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4

Call Stack

(async () => { … })

new *Promise(*
(res) => { … }

Web API

Event Loop

Macrotask Queue

Microtask Queue

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5

Call Stack

console.log(5);

(async () => { … })

new *Promise(*
(res) => { … }

Web API

Event Loop

Macrotask Queue

Microtask Queue

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5

Call Stack

asyncFunction()

(async () => { … })

new *Promise(*
(res) => { … }

Web API

Event Loop

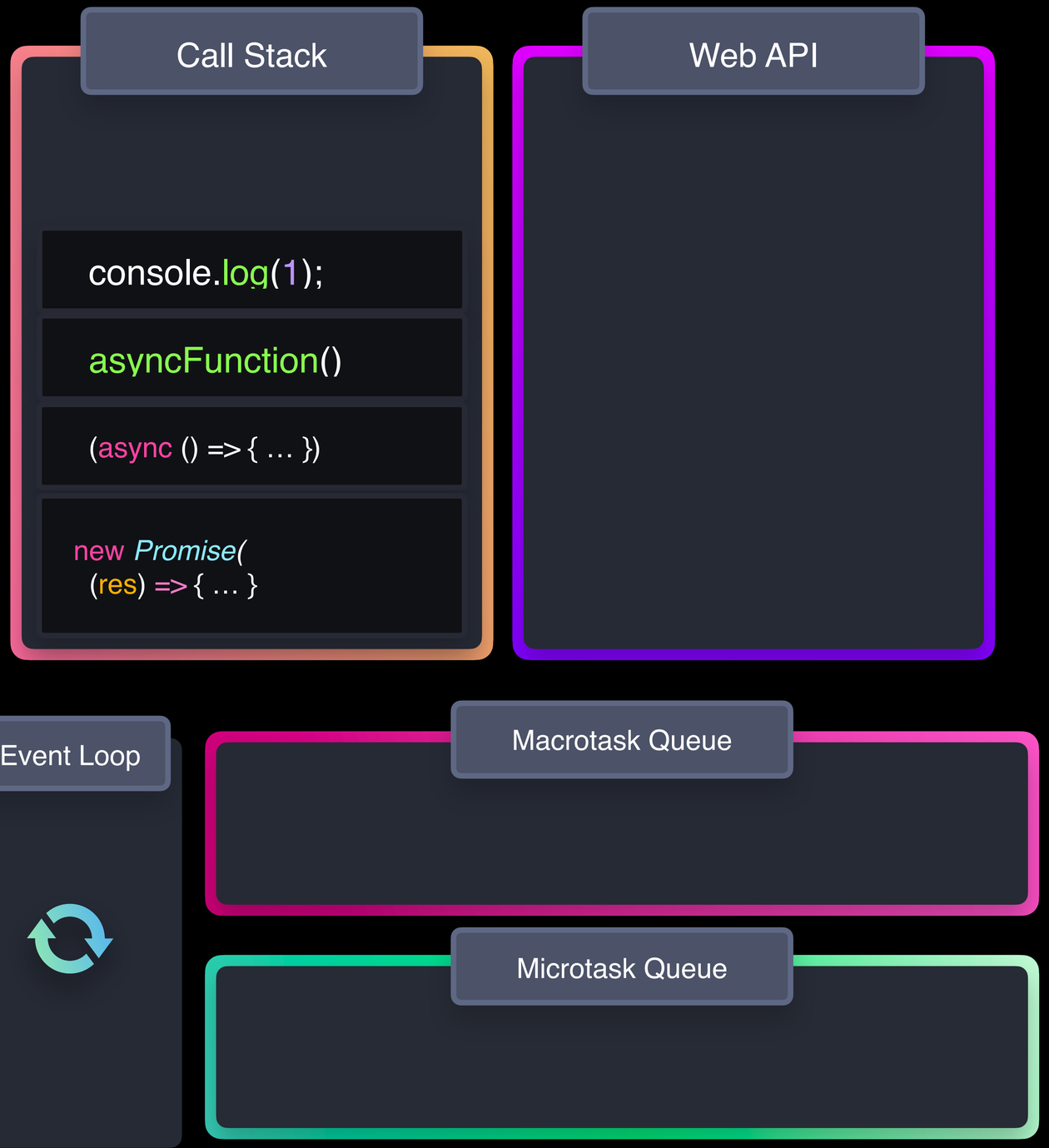
Macrotask Queue

Microtask Queue

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

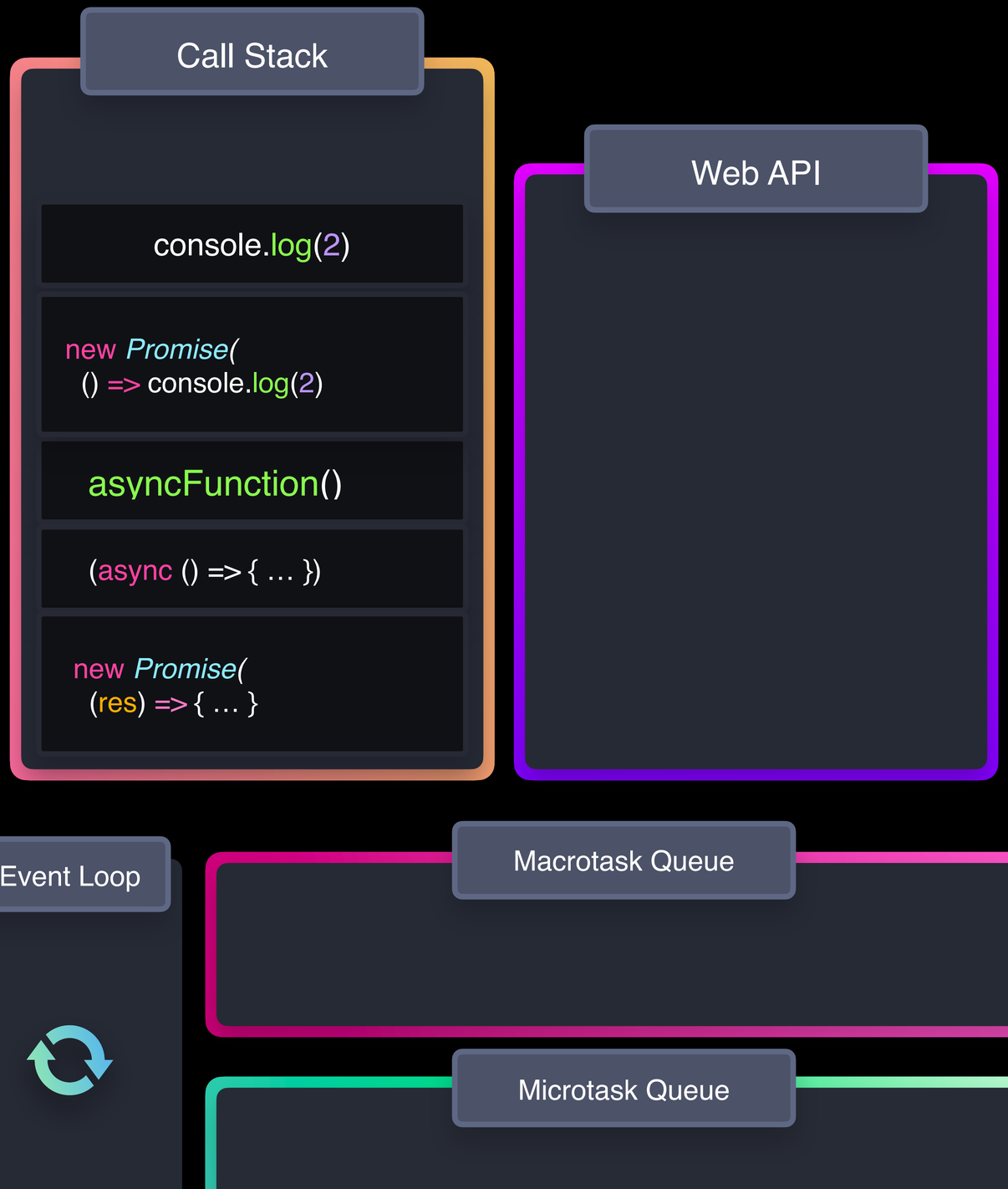
```
4
5
1
```



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

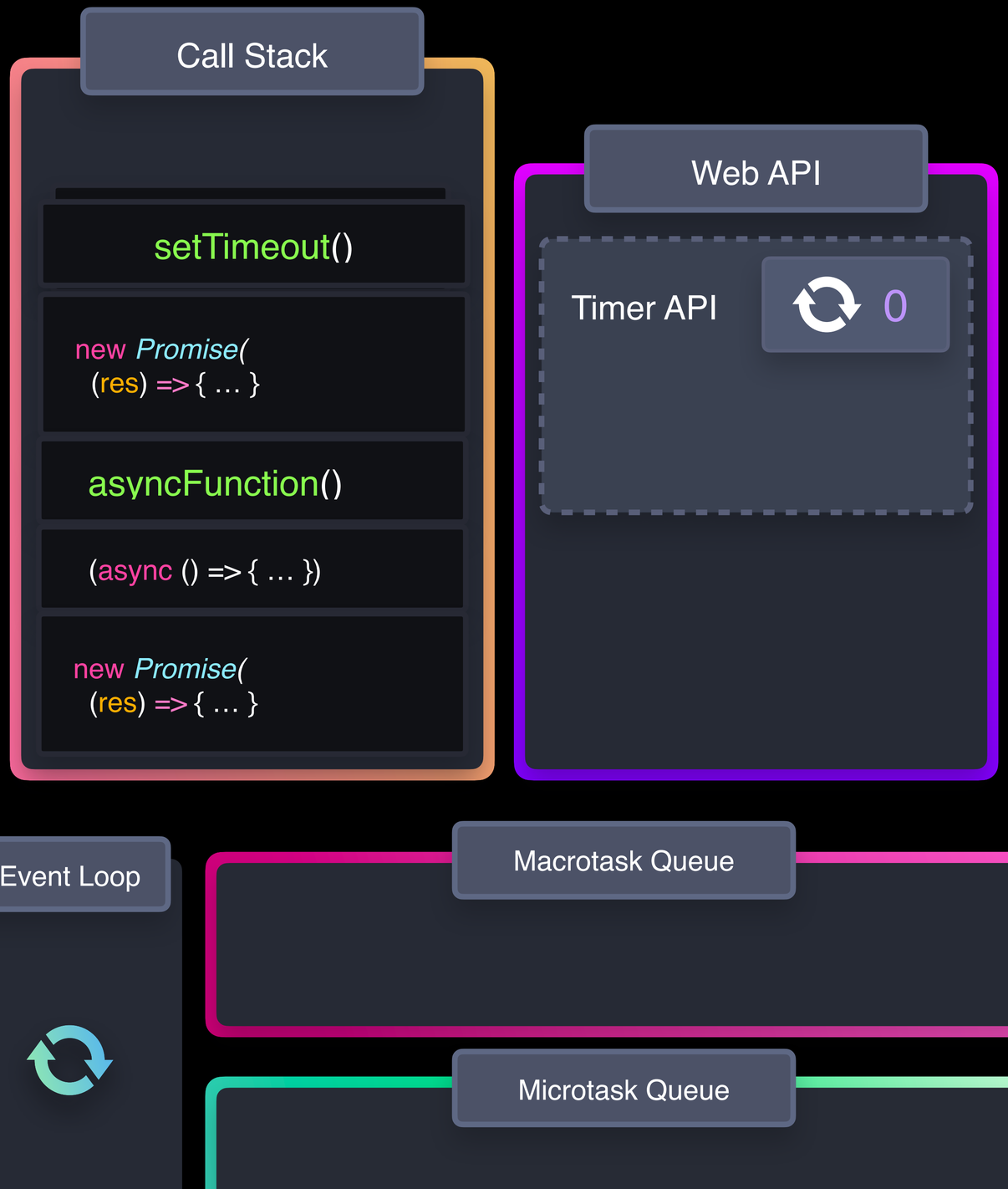
```
4
5
1
2
```



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

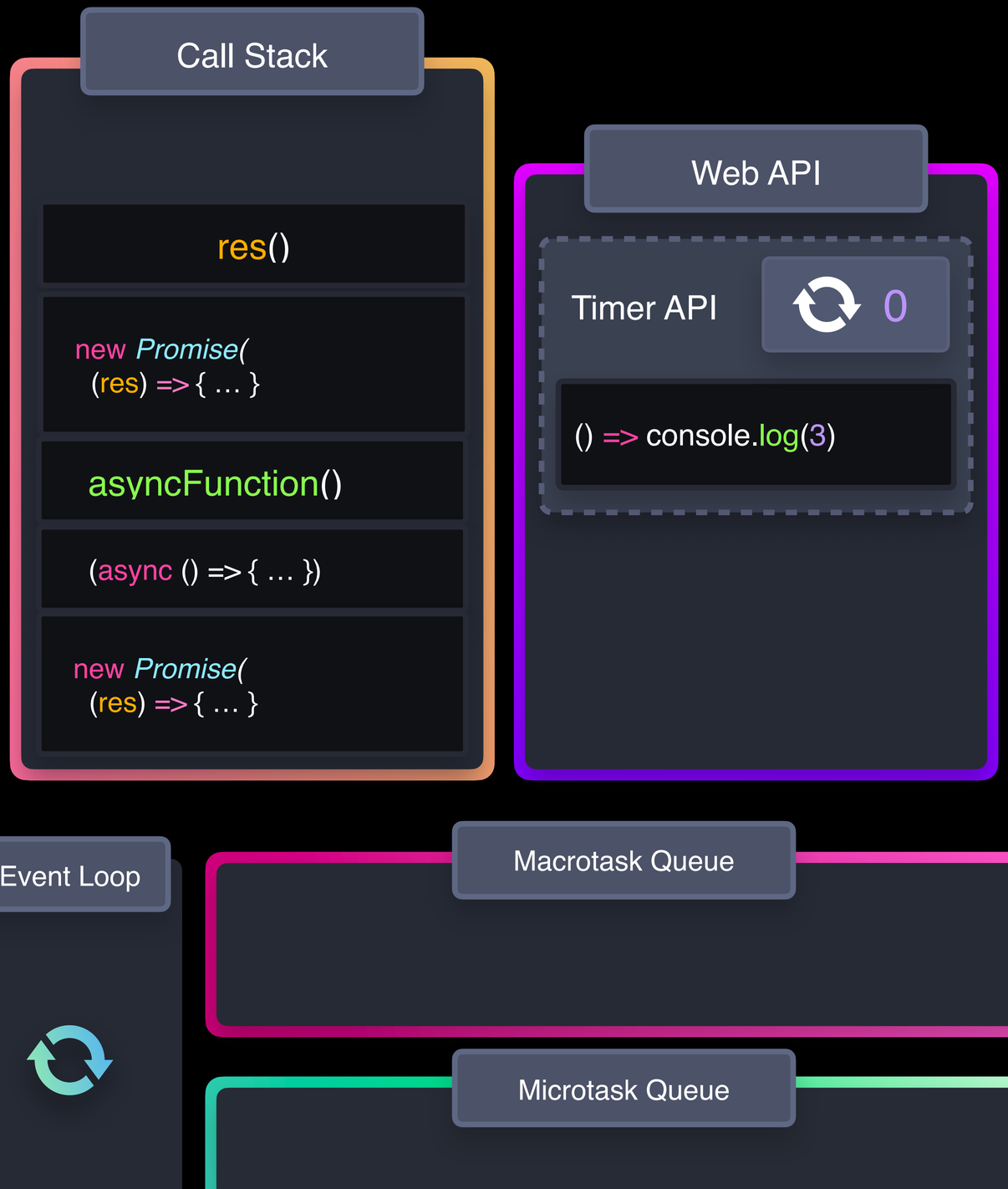
```
4
5
1
2
```



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

```
4
5
1
2
```



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2

Call Stack

asyncFunction()

(async () => { … })

new *Promise(*
(res) => { … }

Web API

Event Loop

Macrotask Queue

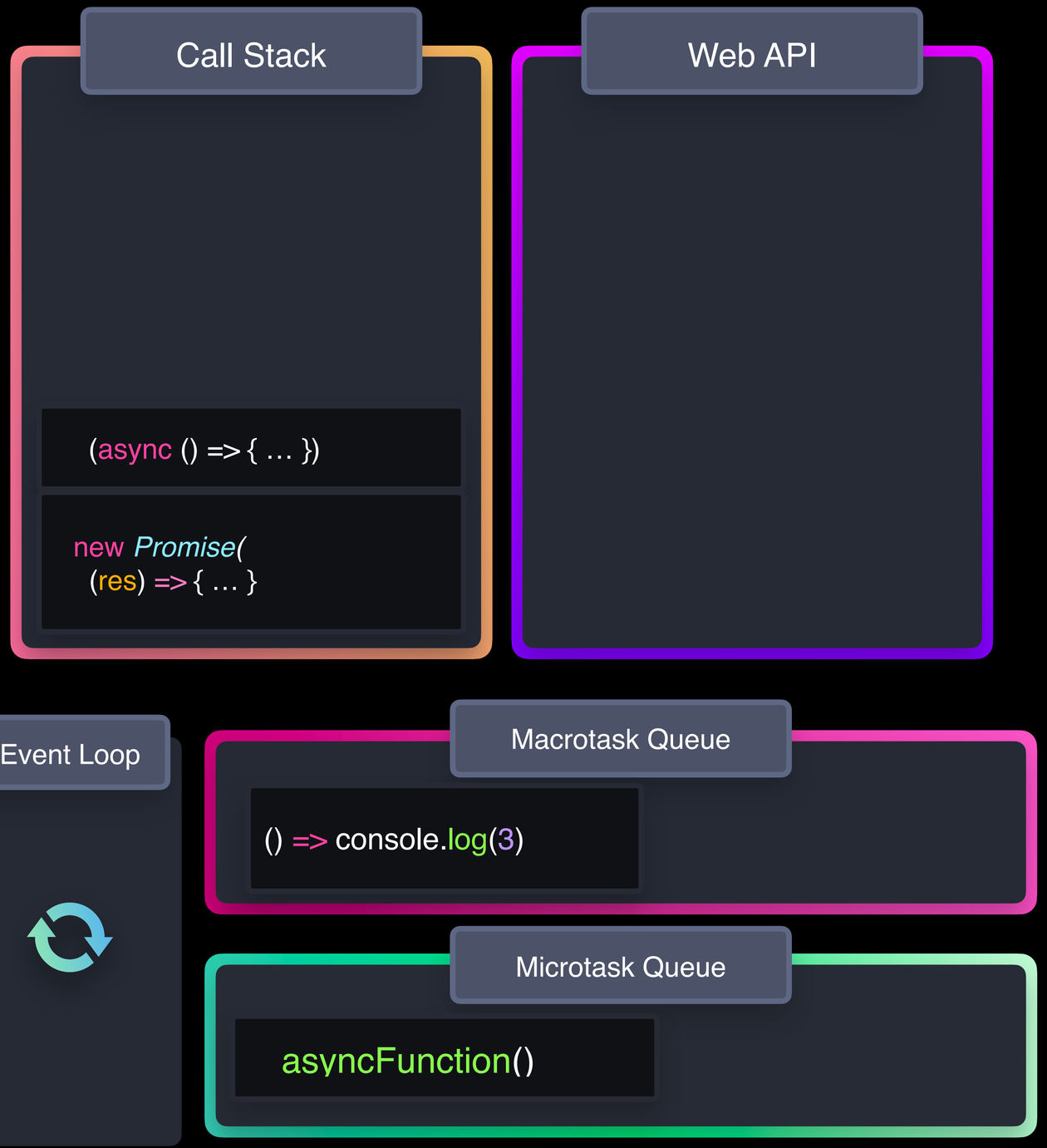
() => console.log(3)

Microtask Queue

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12    (async () => {
13      console.log(5);
14      await asyncFunction();
15      console.log(6);
16    })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

```
4
5
1
2
```



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2

Call Stack

res()

new *Promise(*
(res) => { … }

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

asyncFunction()

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2

Call Stack

then()

Web API

Event Loop

Macrotask Queue

() => console.log(3)

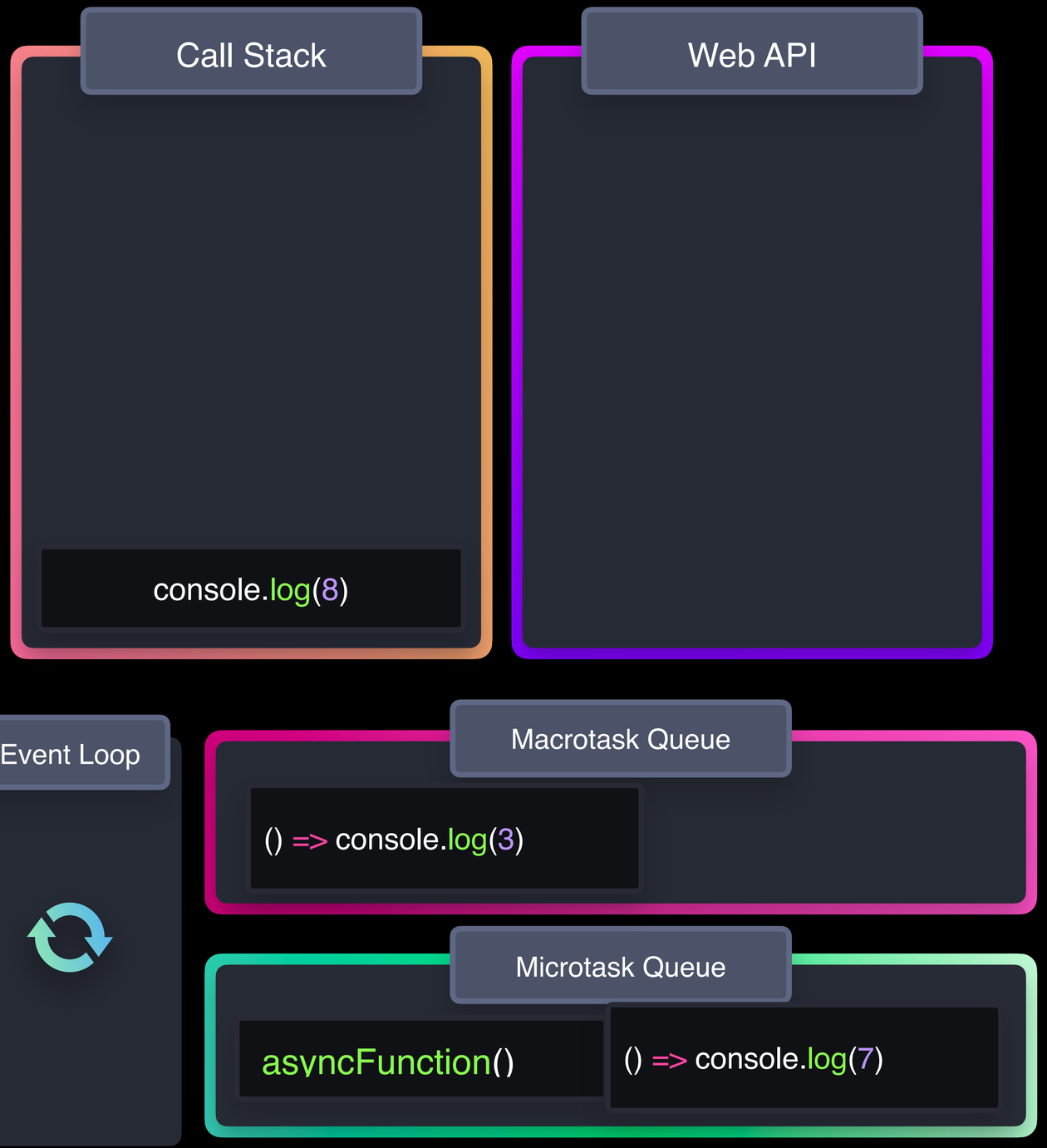
Microtask Queue

asyncFunction()

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2
8



```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2
8

Call Stack

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

asyncFunction()

(async () => { … })

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2
8
7

Call Stack

console.log(7)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

() => console.log(7)

(async () => { … })

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2
8
7
6

Call Stack

console.log(6)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

(async () => { … })

```
1   async function asyncFunction() {
2     console.log(1);
3     new Promise(() => console.log(2));
4     await new Promise((res) => {
5       setTimeout(() => console.log(3), 0);
6       res();
7     });
8   };
9
10  new Promise((res) => {
11    console.log(4);
12      (async () => {
13        console.log(5);
14        await asyncFunction();
15        console.log(6);
16      })();
17    res();
18  }).then(() => console.log(7));
19
20  console.log(8);
```

console

4
5
1
2
8
7
6
3

Call Stack

console.log(3)

Web API

Event Loop

Macrotask Queue

() => console.log(3)

Microtask Queue

Question 4

Choose the correct logs

```
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results = await Promise.all([
        asyncFunc(), promise
  ]);

  return results;
})();

console.log("script")
```

A	promise	async body	script	queueMicrotask	
B	promise	async body	script	then	queueMicrotask
C	script	promise	async body	then	queueMicrotask
D	script	async body	promise	then	queueMicrotask
E	promise	script	async body	queueMicrotask	then

Question 4

Choose the correct logs

```
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results = await Promise.all([
        asyncFunc(), promise
  ]);

  return results;
})();

console.log("script")
```

A promise | async body | script | queueMicrotask

B promise | async body | script | then | queueMicrotask

C script | promise | async body | then | queueMicrotask

D script | async body | promise | then | queueMicrotask

E promise | script | async body | queueMicrotask | then

```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

Call Stack

Web API

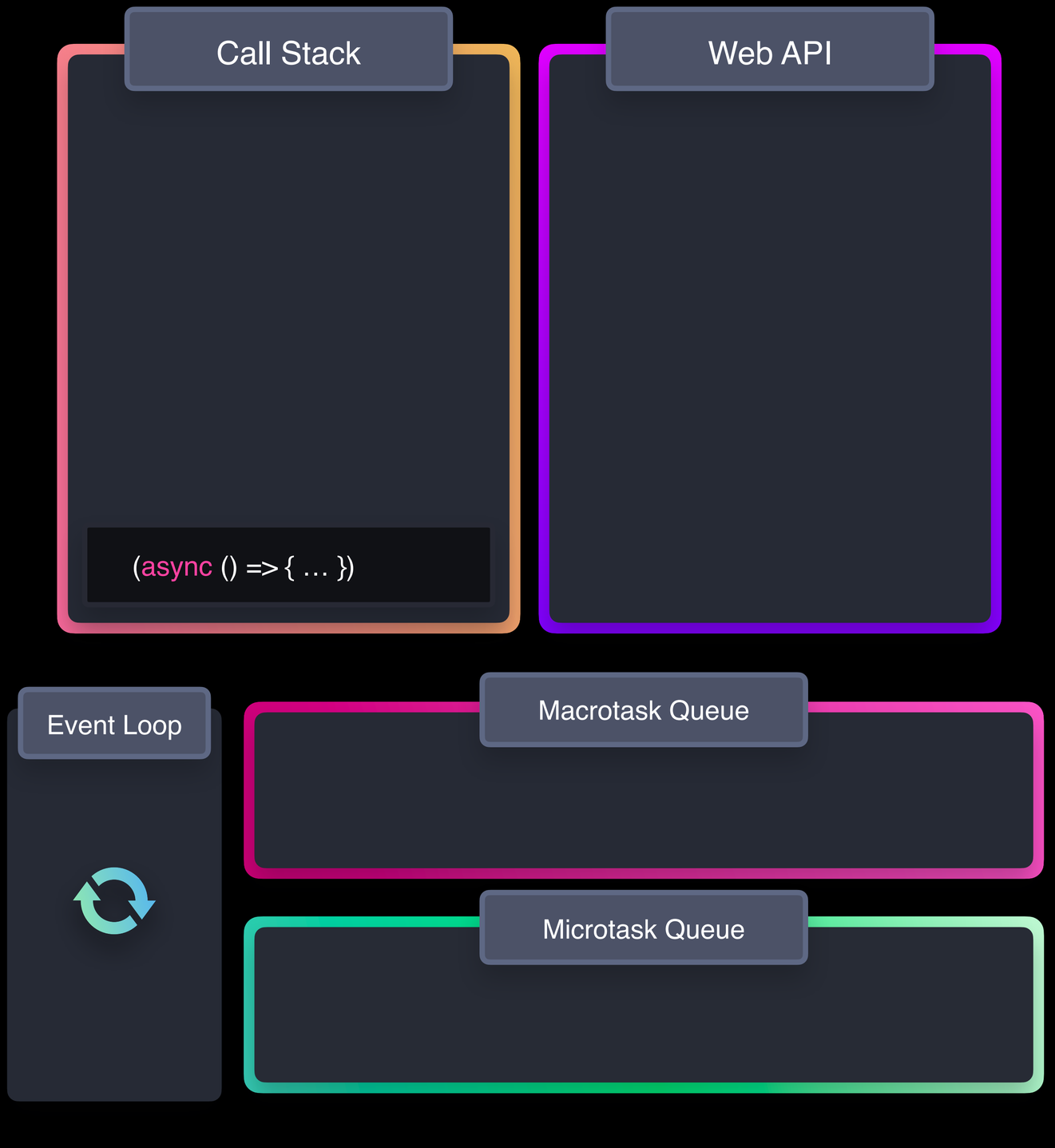
Event Loop

Macrotask Queue

Microtask Queue

```
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

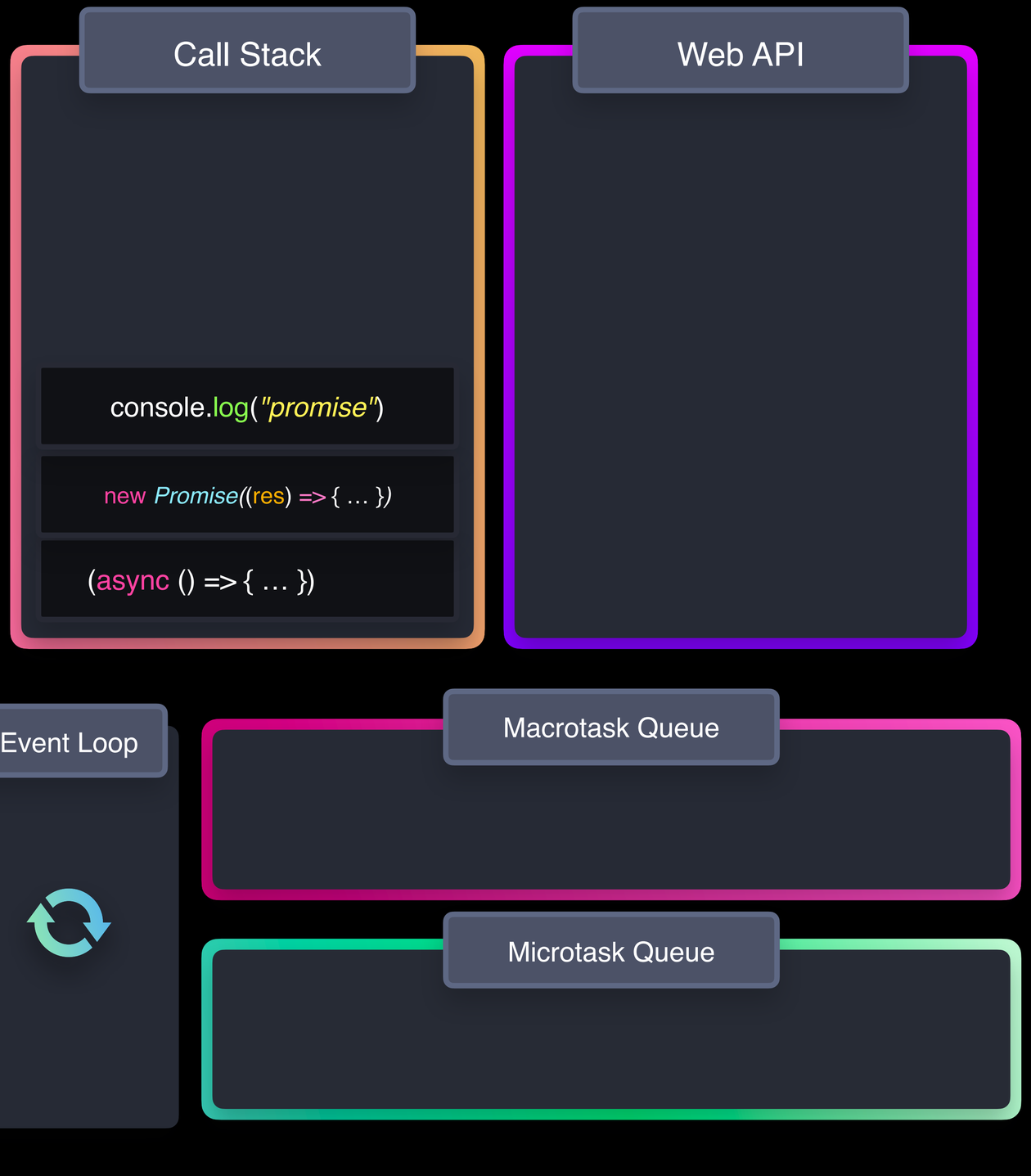
console



```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

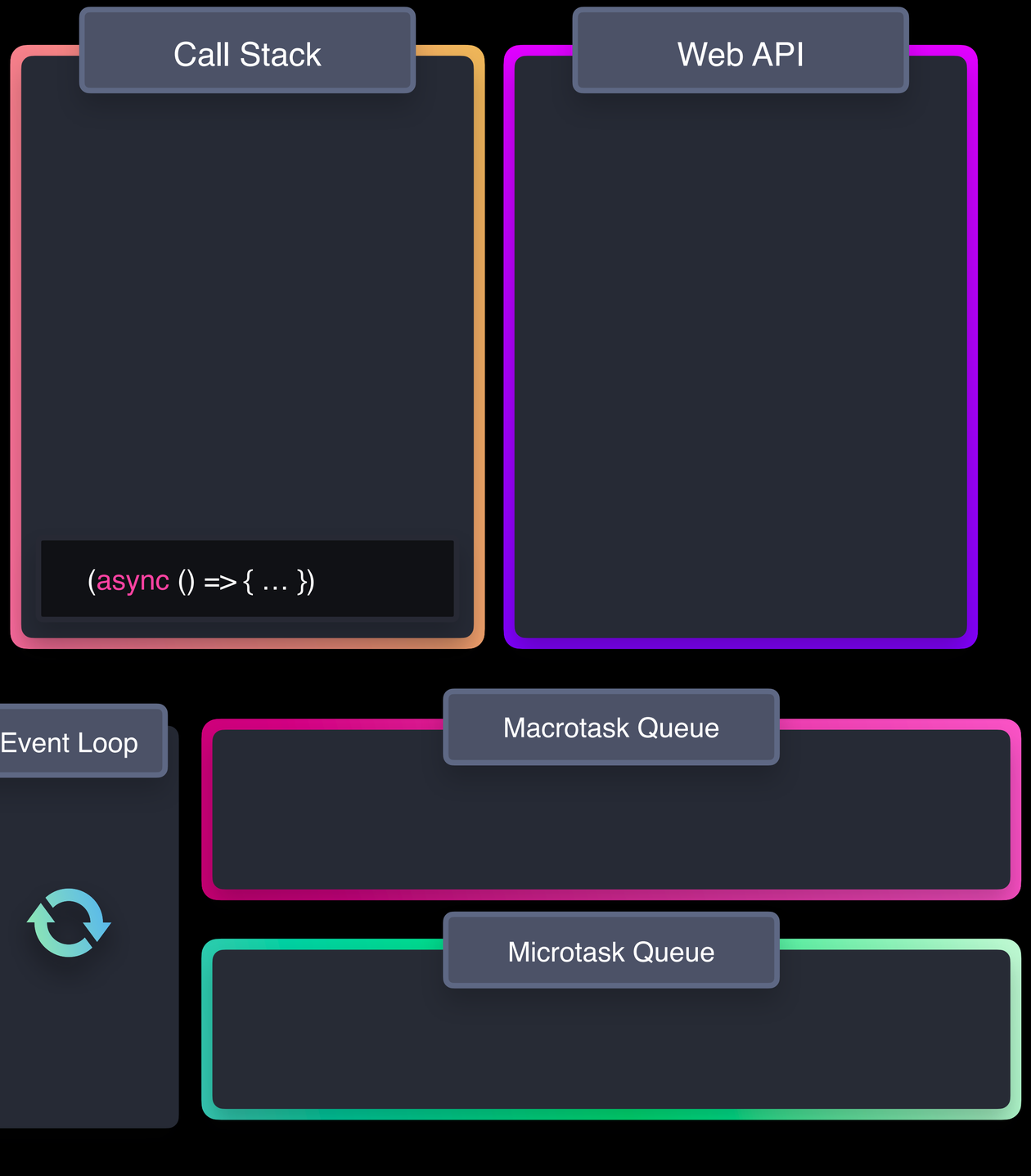
"promise"



```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

"promise"

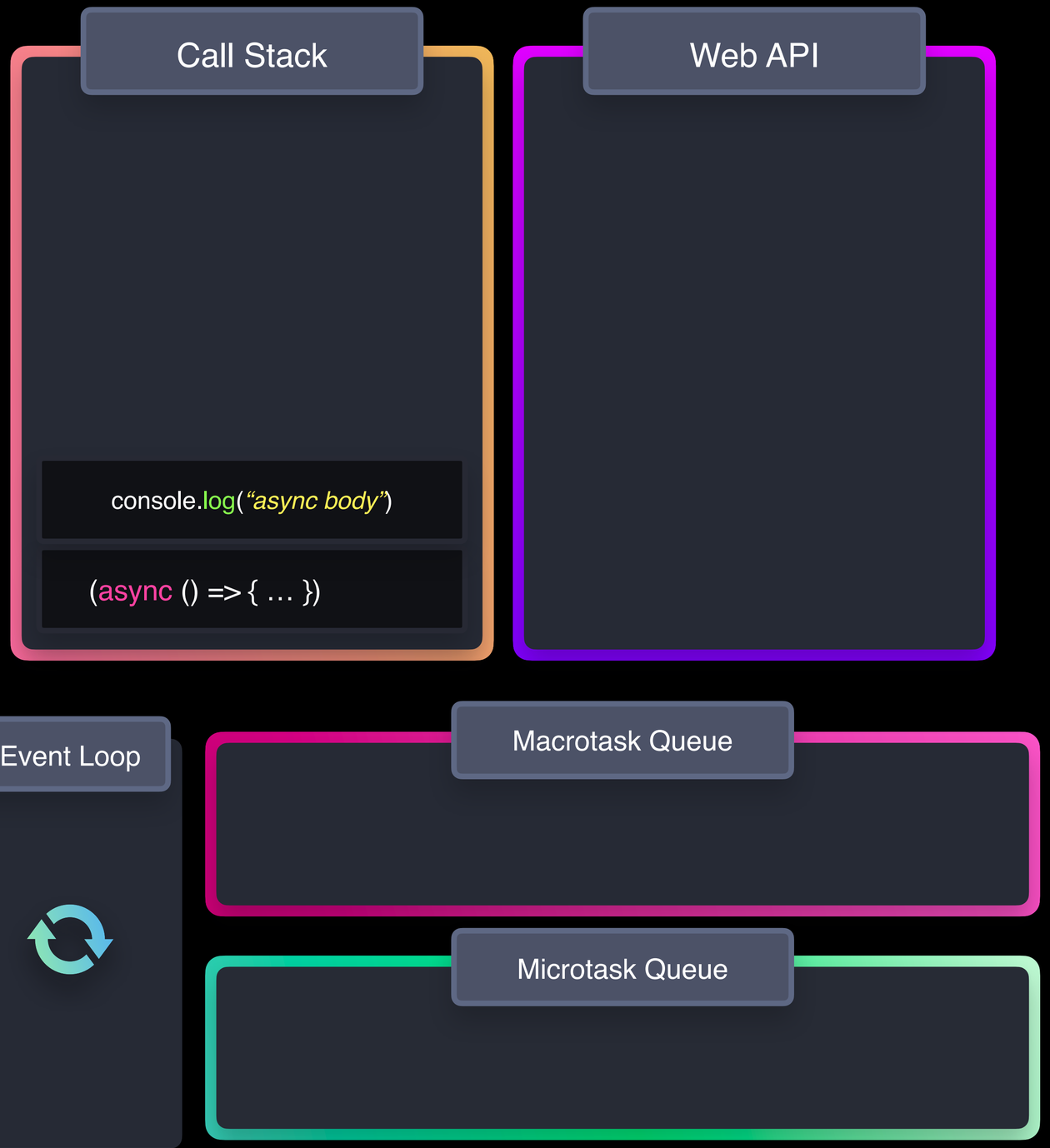


```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

"promise"

"async body"

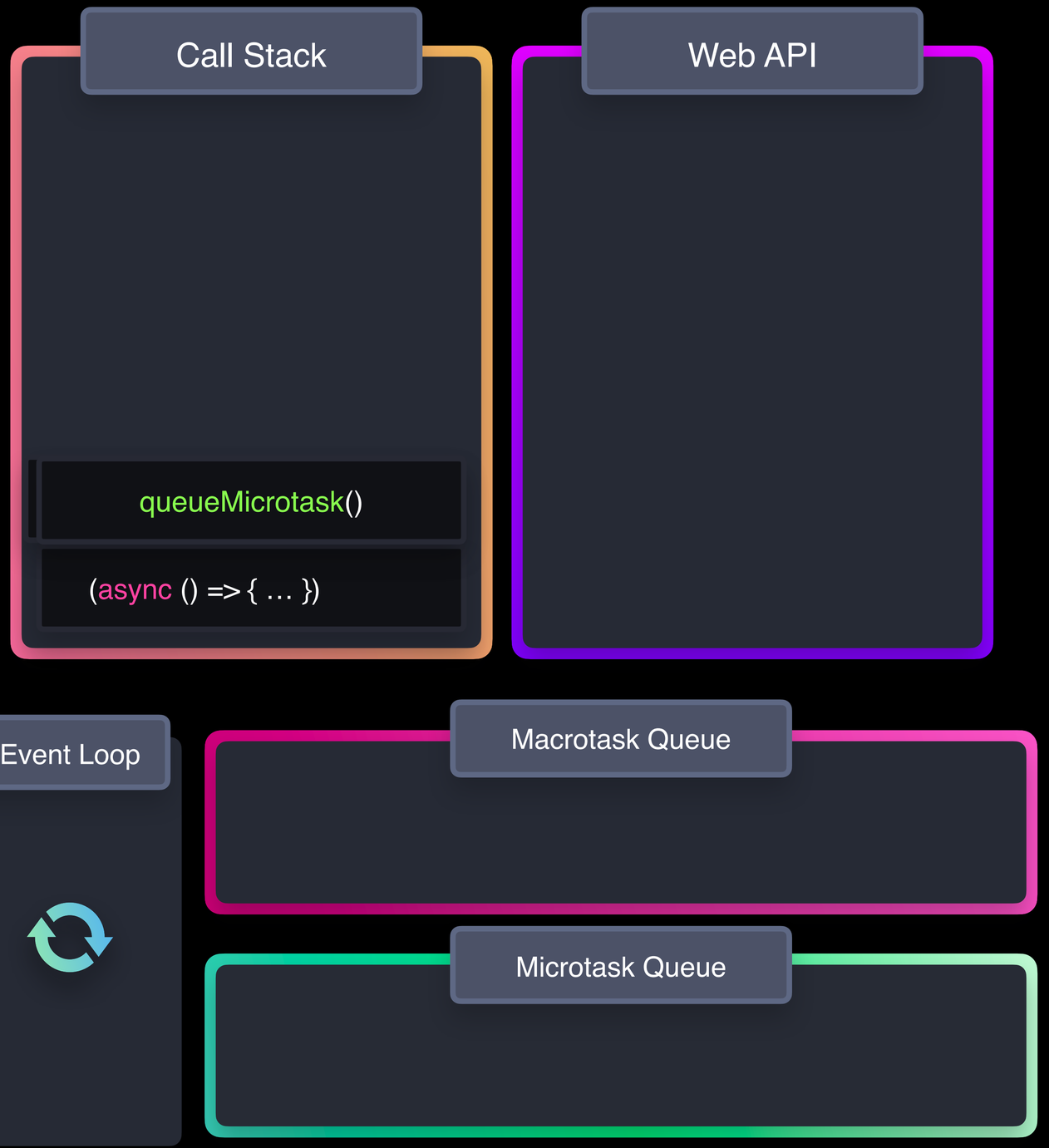


```
1  (async () => {
2    const asyncFunc = async () => "asyncFunc";
3
4    const promise = new Promise(res => {
5      console.log("promise")
6    }).then(() => console.log("then"));
7
8    console.log("async body");
9
10   queueMicrotask(() => {
11     console.log("queueMicrotask")
12   });
13
14   const results =
15           await Promise.all([asyncFunc(), promise]);
16
17   return results;
18 })();
19
20 console.log("script")
```

console

"promise"

"async body"

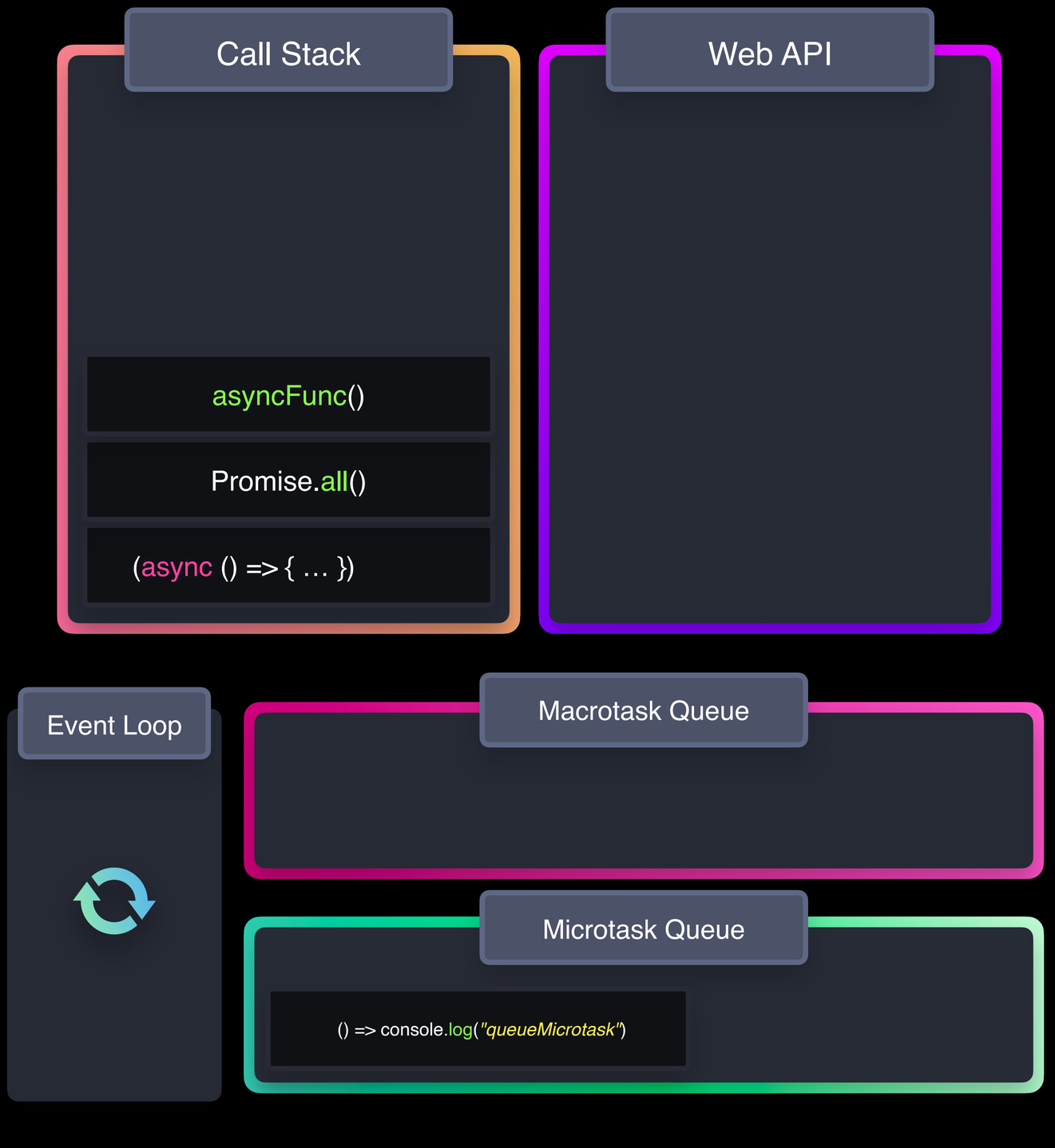


```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

"promise"

"async body"

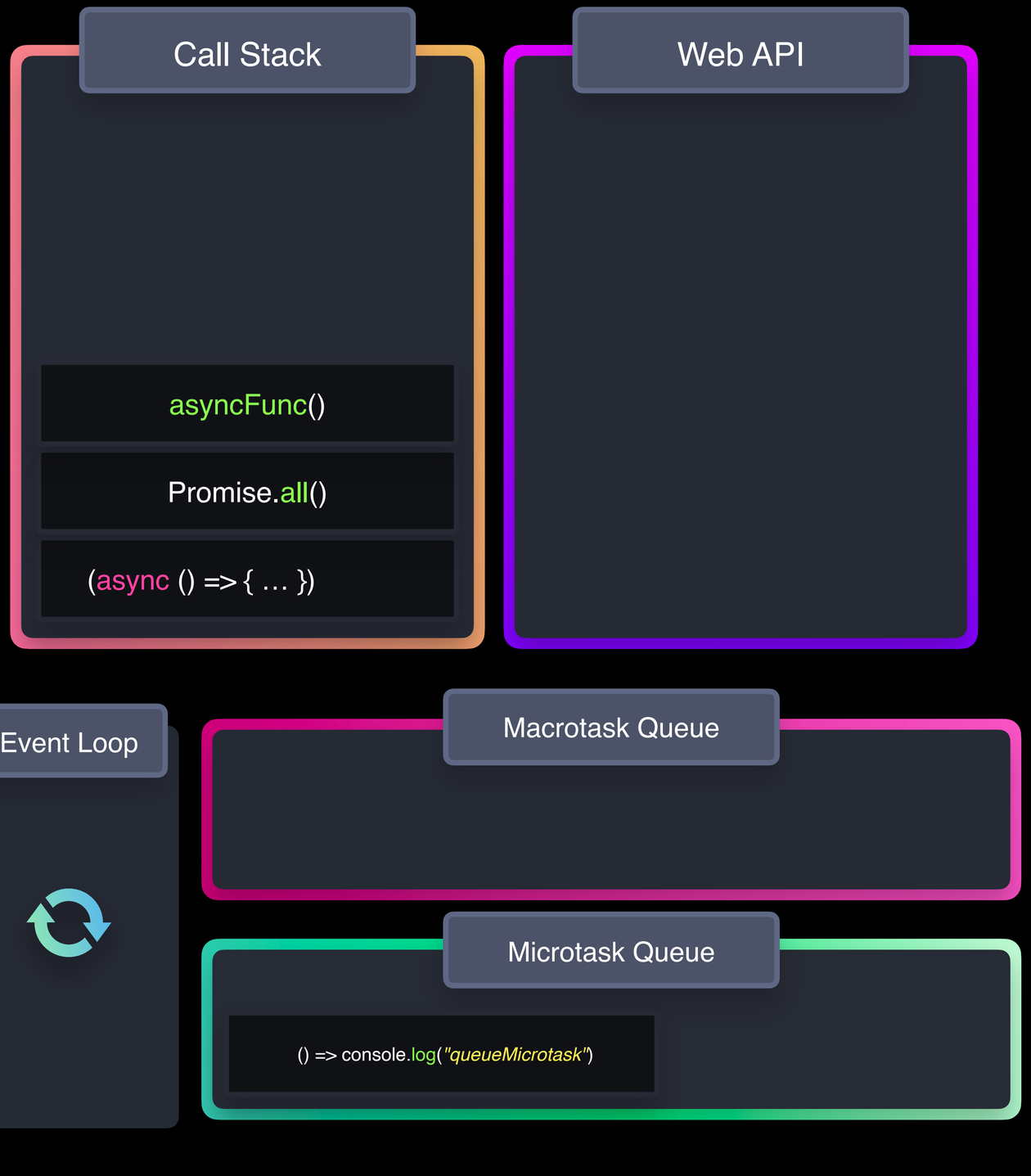


```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

"promise"

"async body"

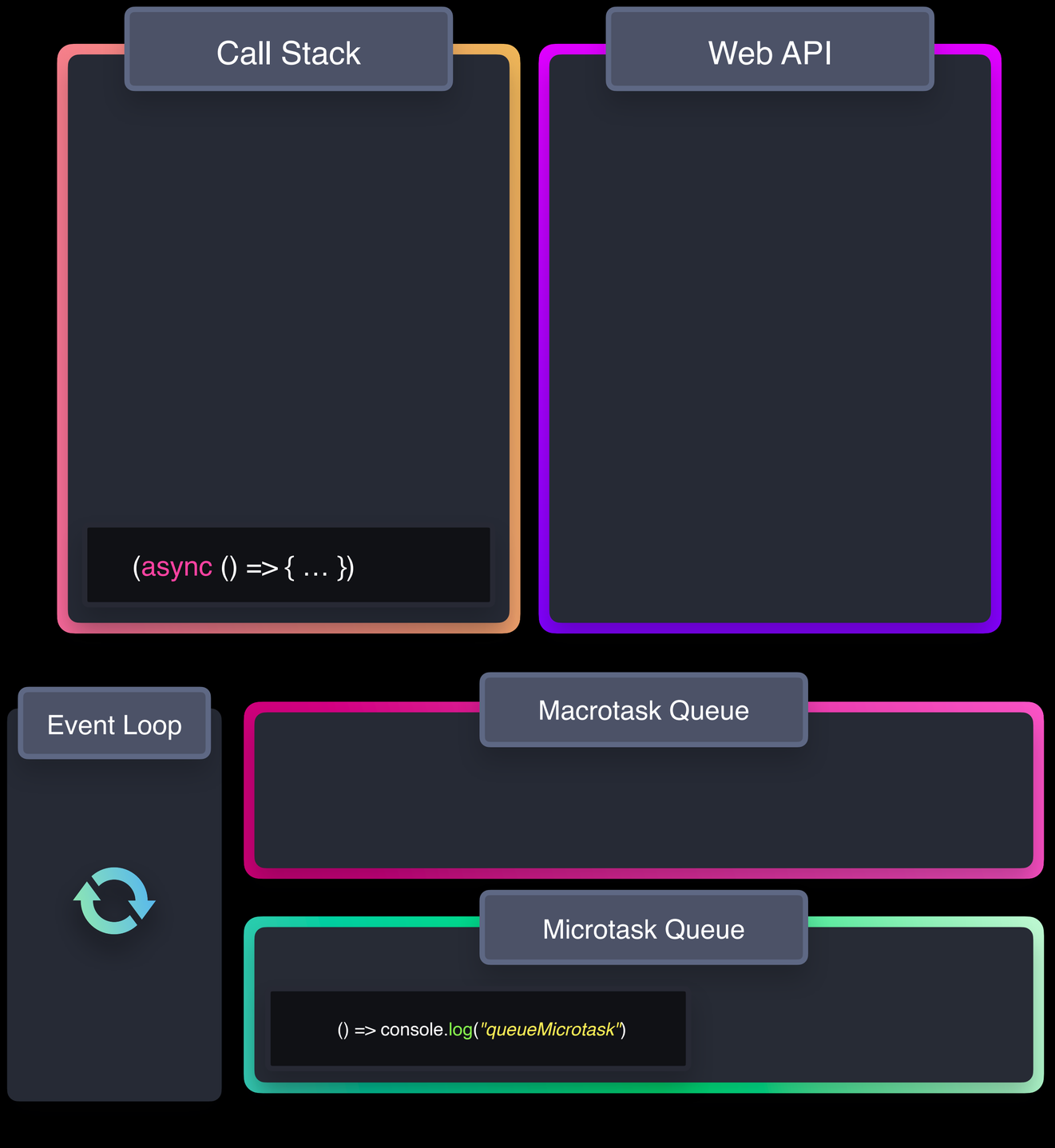


```javascript
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
          await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

"promise"

"async body"



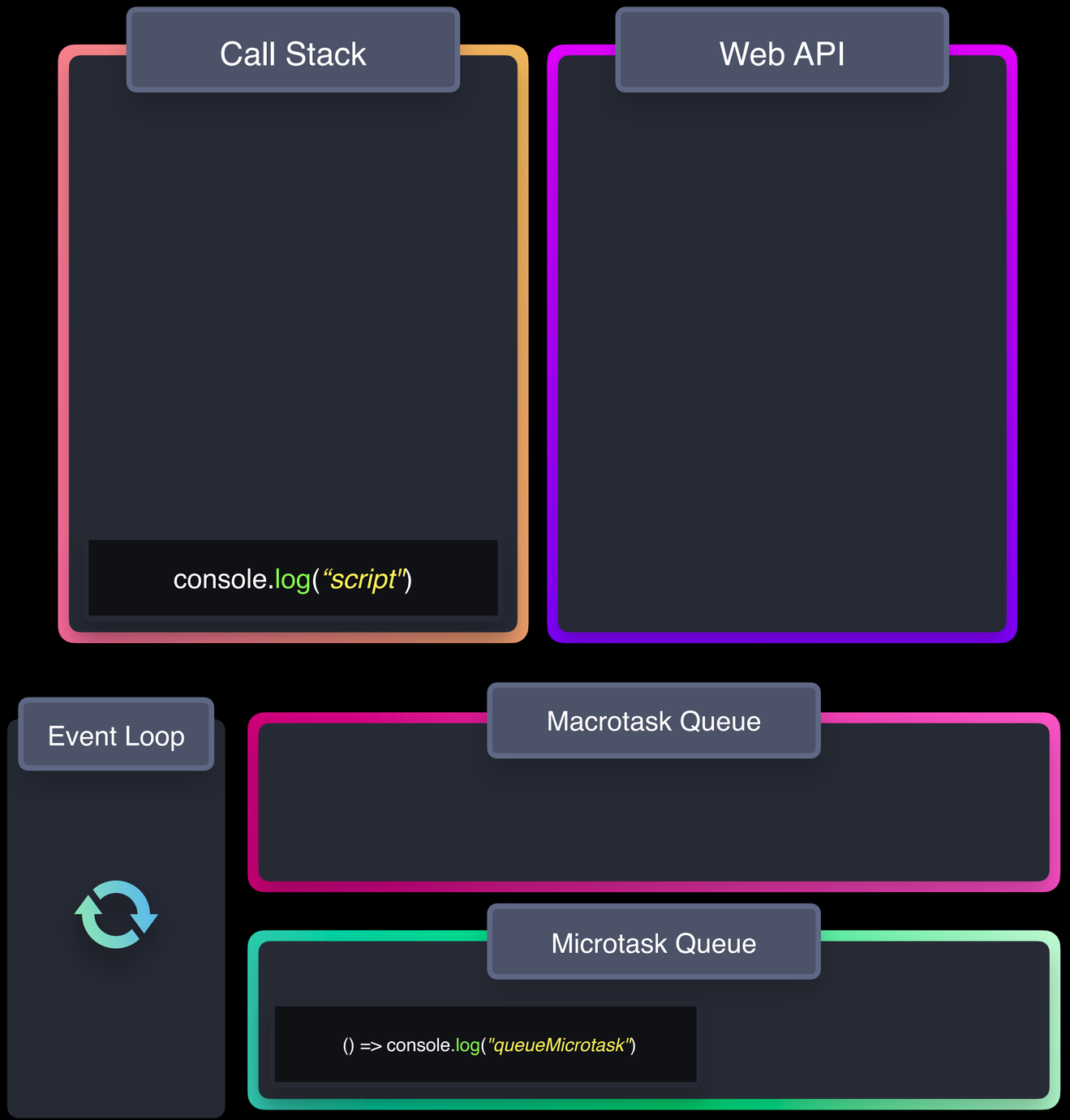
```
(async () => {
  const asyncFunc = async () => "asyncFunc";

  const promise = new Promise(res => {
    console.log("promise")
  }).then(() => console.log("then"));

  console.log("async body");

  queueMicrotask(() => {
    console.log("queueMicrotask")
  });

  const results =
        await Promise.all([asyncFunc(), promise]);

  return results;
})();

console.log("script")
```

console

"promise"

"async body"

"script"



```
1  (async () => {
2    const asyncFunc = async () => "asyncFunc";
3
4    const promise = new Promise(res => {
5      console.log("promise")
6    }).then(() => console.log("then"));
7
8    console.log("async body");
9
10   queueMicrotask(() => {
11     console.log("queueMicrotask")
12   });
13
14   const results =
15           await Promise.all([asyncFunc(), promise]);
16
17   return results;
18 })();
19
20 console.log("script")
```

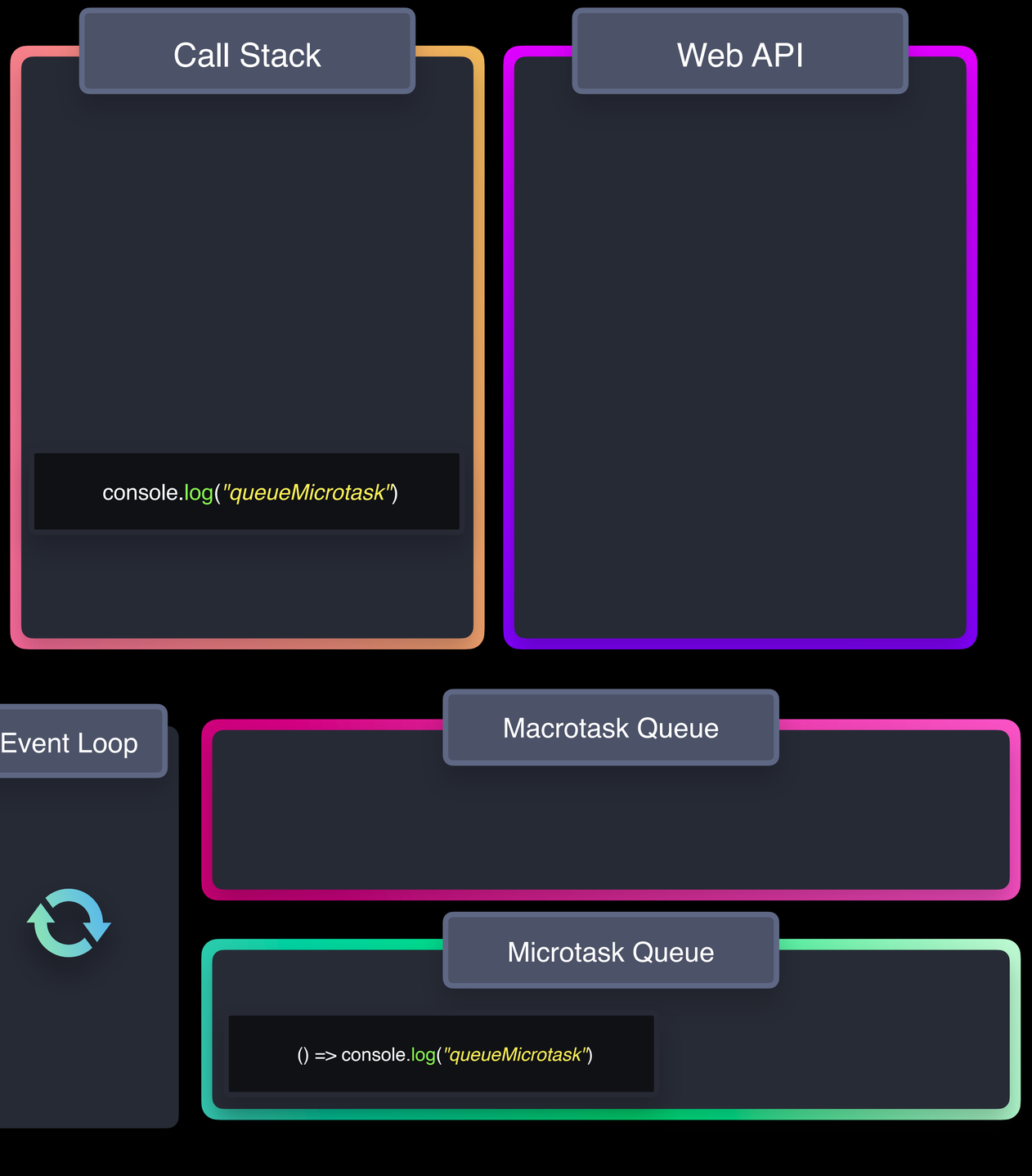
console

"promise"

"async body"

"script"

"queueMicrotask"



```
1  (async () => {
2    const asyncFunc = async () => "asyncFunc";
3
4    const promise = new Promise(res => {
5      console.log("promise")
6    }).then(() => console.log("then"));
7
8    console.log("async body");
9
10   queueMicrotask(() => {
11     console.log("queueMicrotask")
12   });
13
14   const results =
15           await Promise.all([asyncFunc(), promise]);
16
17   return results;
18 })();
19
20 console.log("script")
```

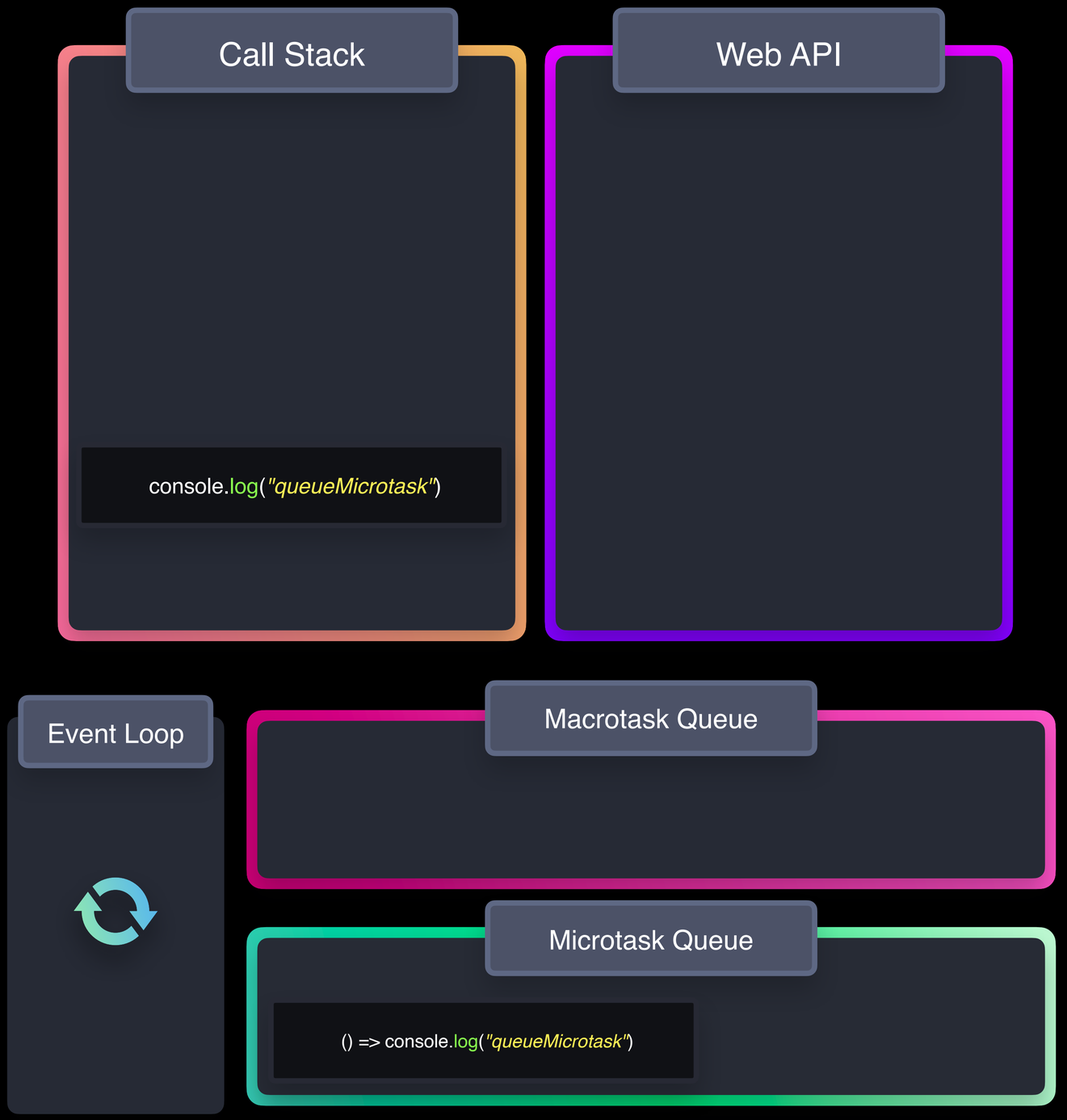
console

"promise"

"async body"

"script"

"queueMicrotask"



Scope & Closures

```
1  const fruit = "lemon";
2
3  {
4     const fruit = "strawberry";
5  }
6
7  function myFunc() {
8    const fruit = "orange";
9  }
```

Global Scope

Block Scope

Function Scope

```
1   function myFunc() {
2       console.log(fruit)
3
4       return function() {
5           console.log(fruit)
6
7           return function() {
8               const fruit = "orange";
9               console.log(fruit)
10          }
11      }
12  }
13
14  console.log(fruit)
```

ReferenceError

ReferenceError

ReferenceError

```
1   function myFunc() {
2       console.log(fruit)
3
4       return function() {
5           const fruit = "orange";
6           console.log(fruit)
7
8           return function() {
9               console.log(fruit)
10          }
11      }
12  }
13
14  console.log(fruit)
```

ReferenceError

ReferenceError

```
function myFunc() {
    const fruit = "orange";
    console.log(fruit)

    return function() {
        console.log(fruit)

        return function() {
            console.log(fruit)
        }
    }
}

console.log(fruit)
```

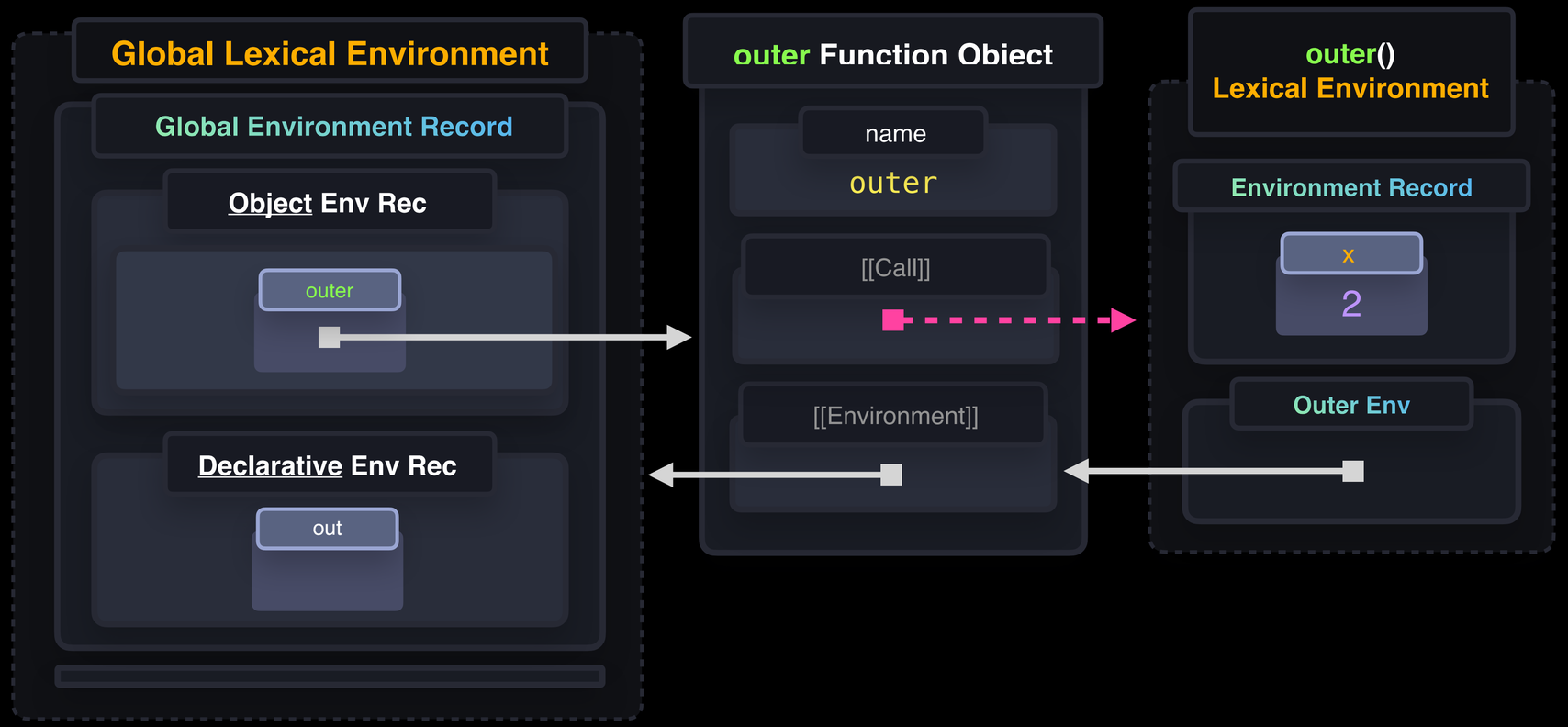
ReferenceError

```
1   const fruit = "orange";
2   function myFunc() {
3       console.log(fruit)
4
5       return function() {
6           console.log(fruit)
7
8           return function() {
9               console.log(fruit)
10          }
11      }
12  }
13
14  console.log(fruit)
```

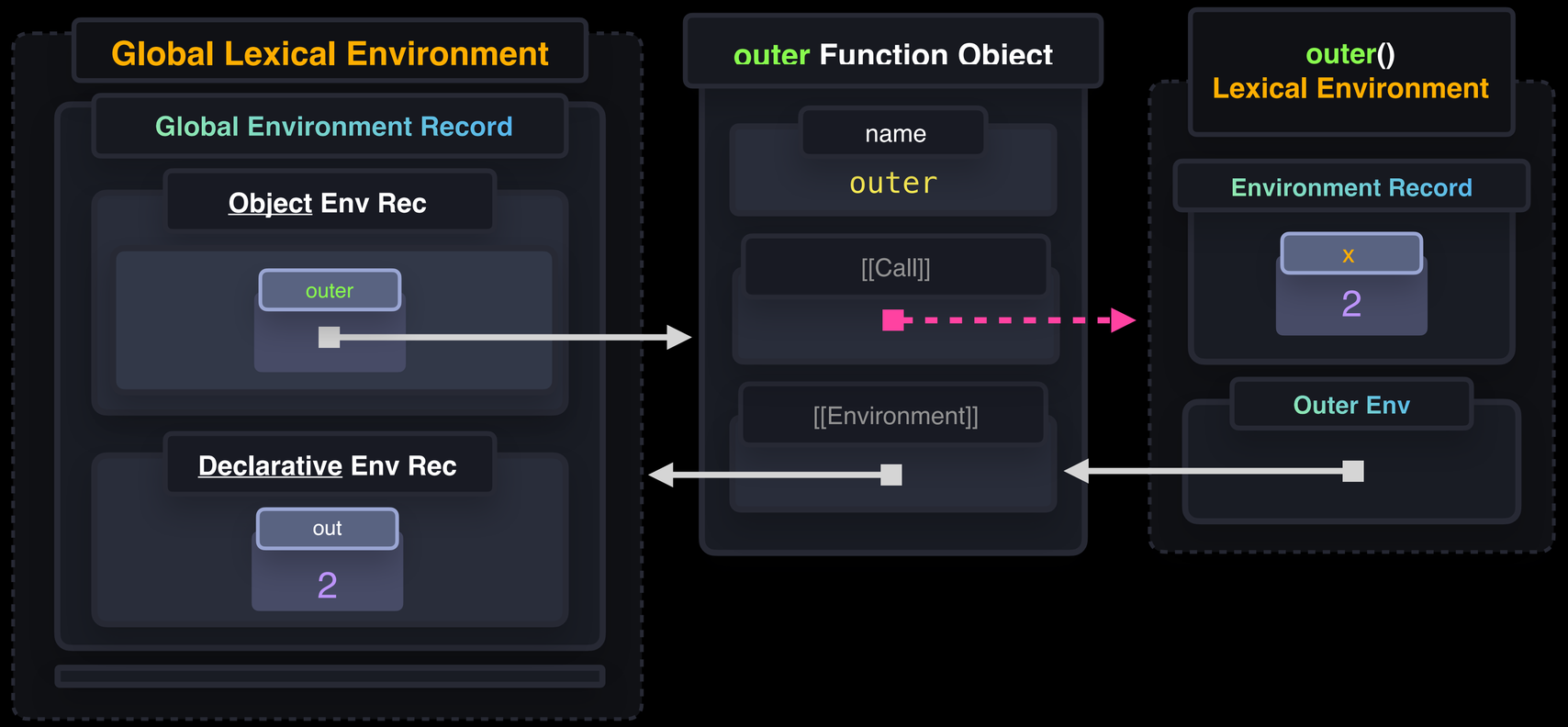
Closures

```
1  function outer(x) {
2    return function inner(y) {
3      return x + y;
4    }
5  }
6
7  const out = outer(2);
8  out(3);
```

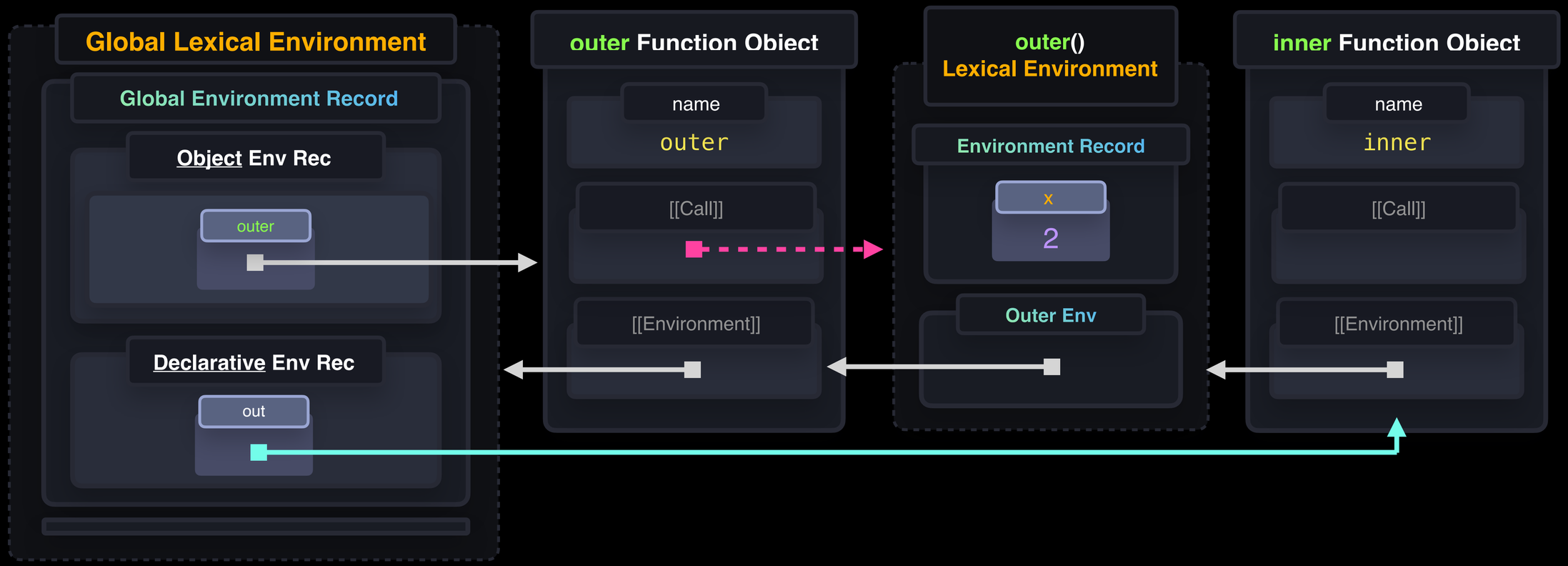
```
1   function outer(x) {
2     return x;
3   }
4
5   const out = outer(2);
```



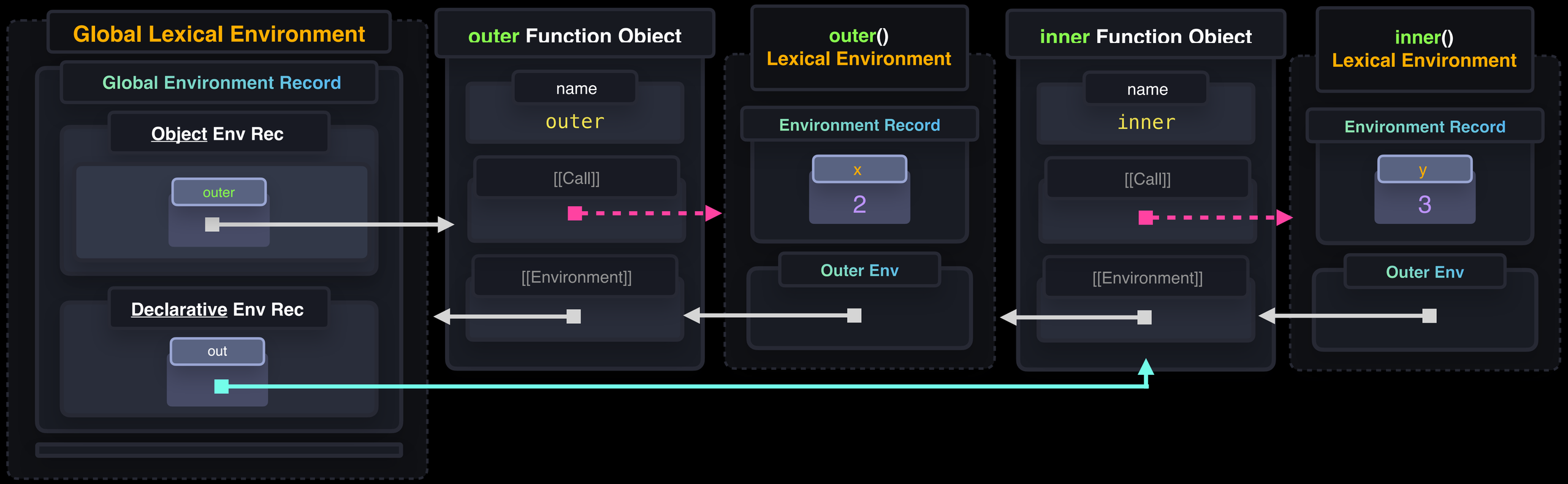
```
1   function outer(x) {
2     return x;
3   }
4
5   const out = outer(2);
```



```
1   function outer(x) {
2     return function inner(y) {
3       return x + y;
4     }
5   }
6
7   const out = outer(2);
8   out(3);
```



```
1  function outer(x) {
2    return function inner(y) {
3      return x + y;
4    }
5  }
6
7  const out = outer(2);
8  out(3);
```



Question 5

What gets logged?

```
1  const outerFunc = () => {
2    let count = 0;
3    return () => ++count;
4  };
5
6  const counter = outerFunc();
7  console.log(counter());
8  console.log(counter());
```

A 0 1

B 1 2

C 1 1

D 0 0

Question 5

What gets logged?

```
1  const outerFunc = () => {
2    let count = 0;
3    return () => ++count;
4  };
5
6  const counter = outerFunc();
7  console.log(counter());
8  console.log(counter());
```

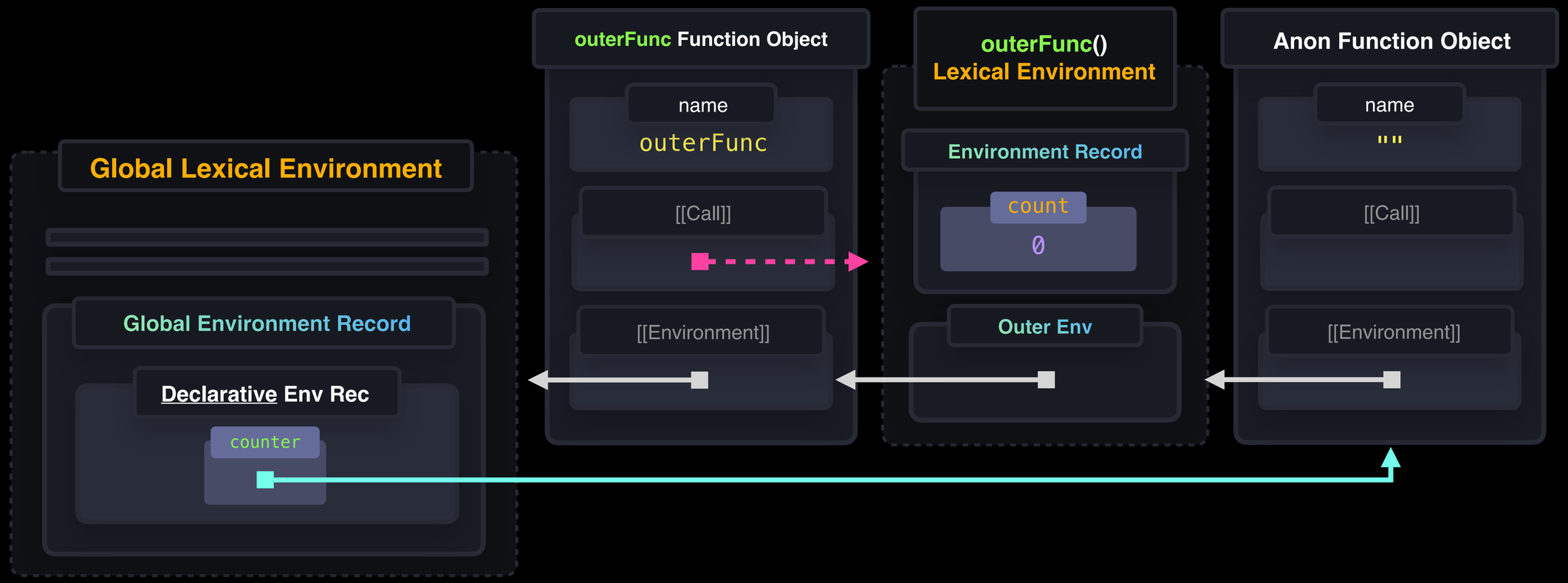
A 0 1

B 1 2

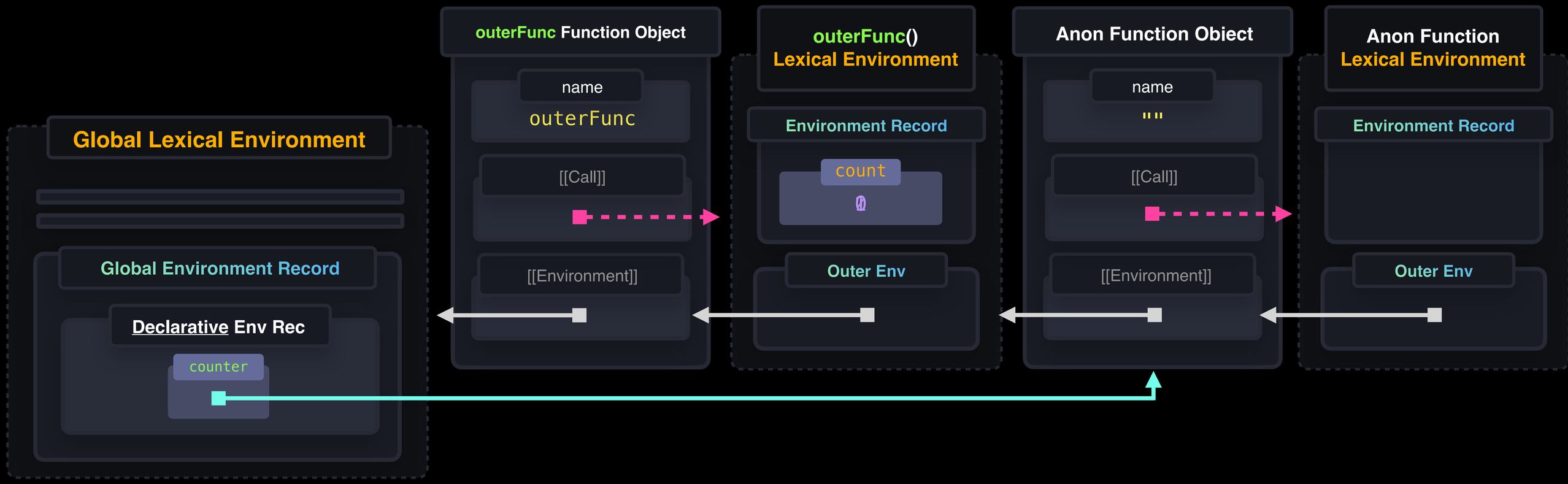
C 1 1

D 0 0

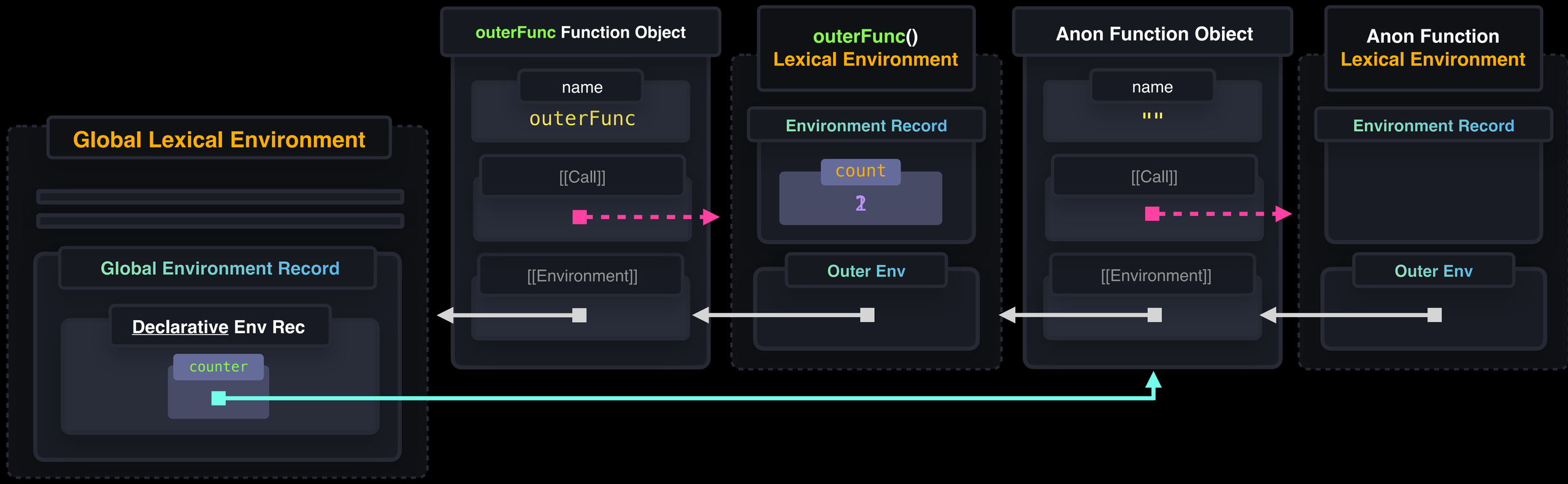
```
1  const outerFunc = () => {
2    let count = 0;
3    return () => ++count;
4  };
5
6  const counter = outerFunc();
7  console.log(counter());
8  console.log(counter());
```



```
1   const outerFunc = () => {
2     let count = 0;
3     return () => ++count;
4   };
5
6   const counter = outerFunc();
7   console.log(counter());
8   console.log(counter());
```



```
1  const outerFunc = () => {
2    let count = 0;
3    return () => ++count;
4  };
5
6  const counter = outerFunc();
7  console.log(counter());
8  console.log(counter());
```



Question 6

What gets logged?

```
1  function createCounter() {
2    let globalCount = 0;
3
4    function incrementCount() {
5        let incrementedValue = ++globalCount;
6        return incrementedValue;
7      }
8
9    return { incrementCount }
10 };
11
12 const counter = createCounter();
13 console.log(counter.incrementCount());
14 console.log(counter.incrementCount());
15 console.log(createCounter().incrementCount());
```

A 0 1 0

B 1 1 1

C 0 1 2

D 1 2 1

E 1 2 3

Question 6

What gets logged?

```
1  function createCounter() {
2    let globalCount = 0;
3
4    function incrementCount() {
5        let incrementedValue = ++globalCount;
6        return incrementedValue;
7      }
8
9    return { incrementCount }
10 };
11
12 const counter = createCounter();
13 console.log(counter.incrementCount());
14 console.log(counter.incrementCount());
15 console.log(createCounter().incrementCount());
```

A 0 1 0

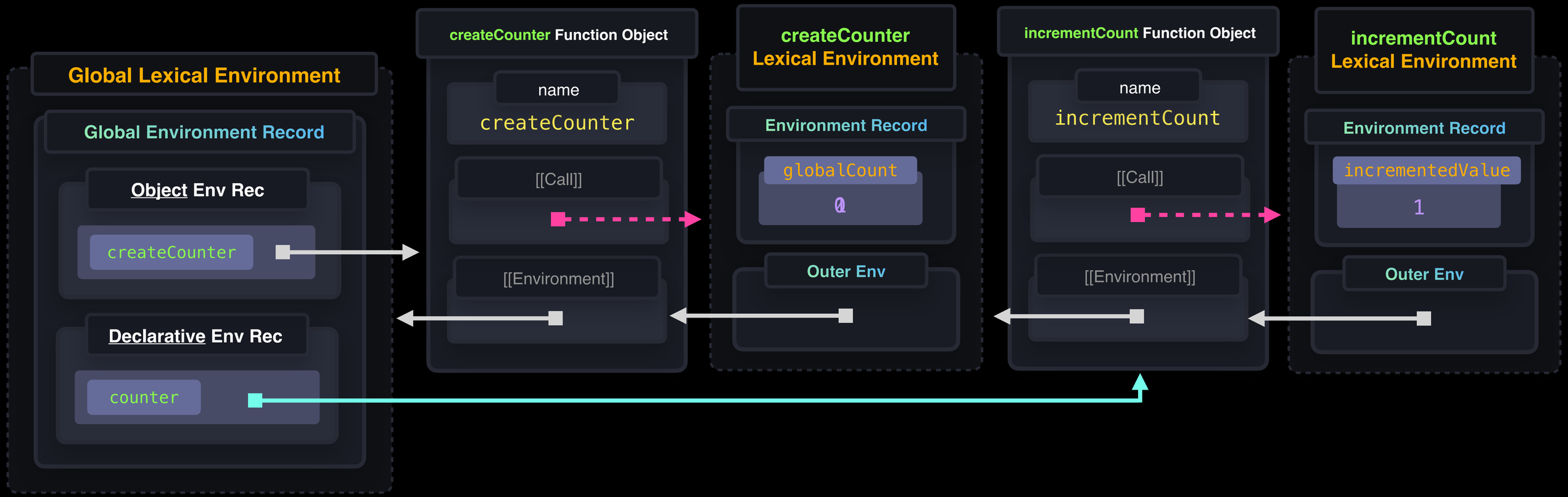
B 1 1 1

C 0 1 2

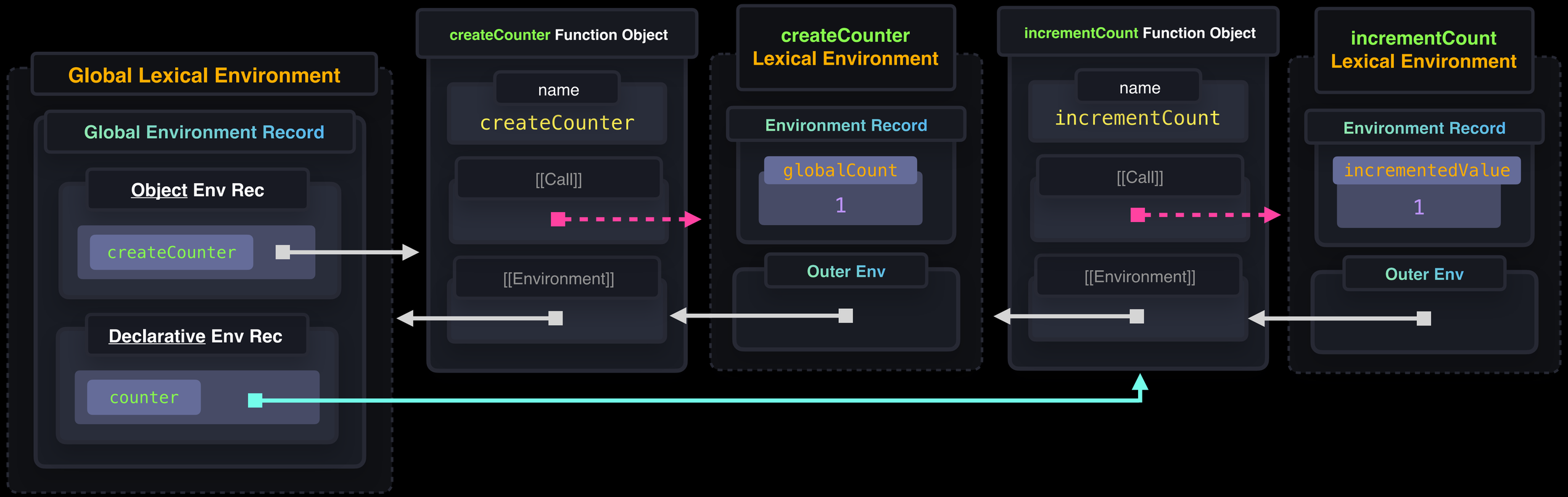
D 1 2 1

E 1 2 3

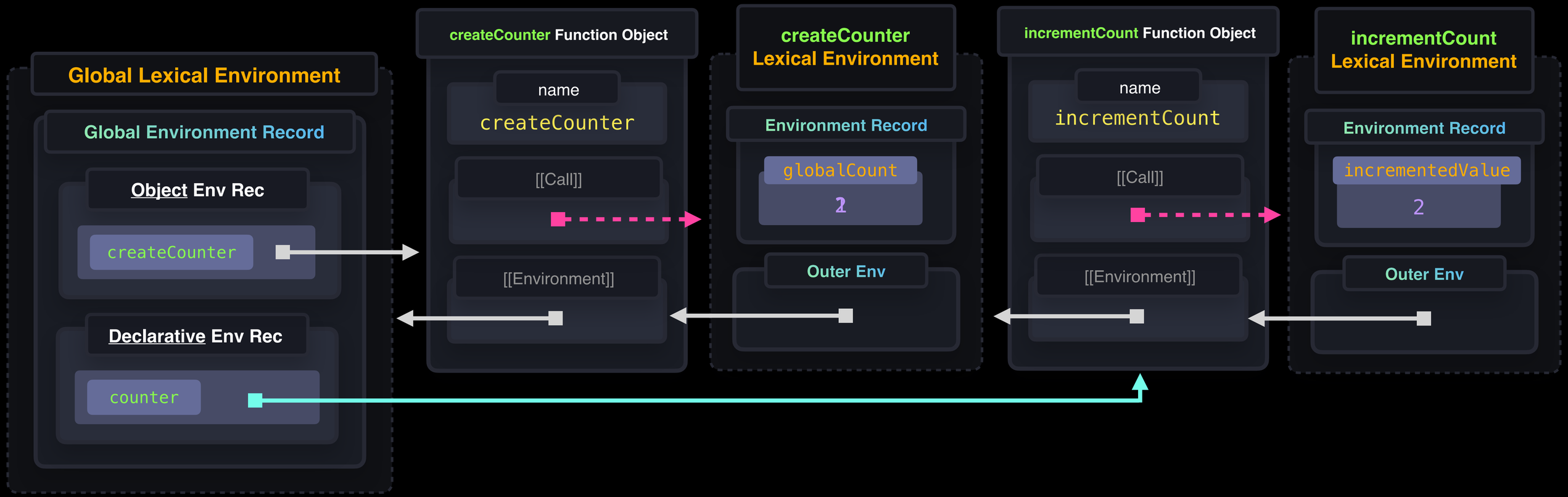
```
1  function createCounter() {
2    let globalCount = 0;
3
4    function incrementCount() {
5        let incrementedValue = ++globalCount;
6        return incrementedValue;
7      }
8
9    return { incrementCount }
10 };
11 const counter = createCounter();
12 console.log(counter.incrementCount());
13 console.log(counter.incrementCount());
14 console.log(createCounter().incrementCount());
```



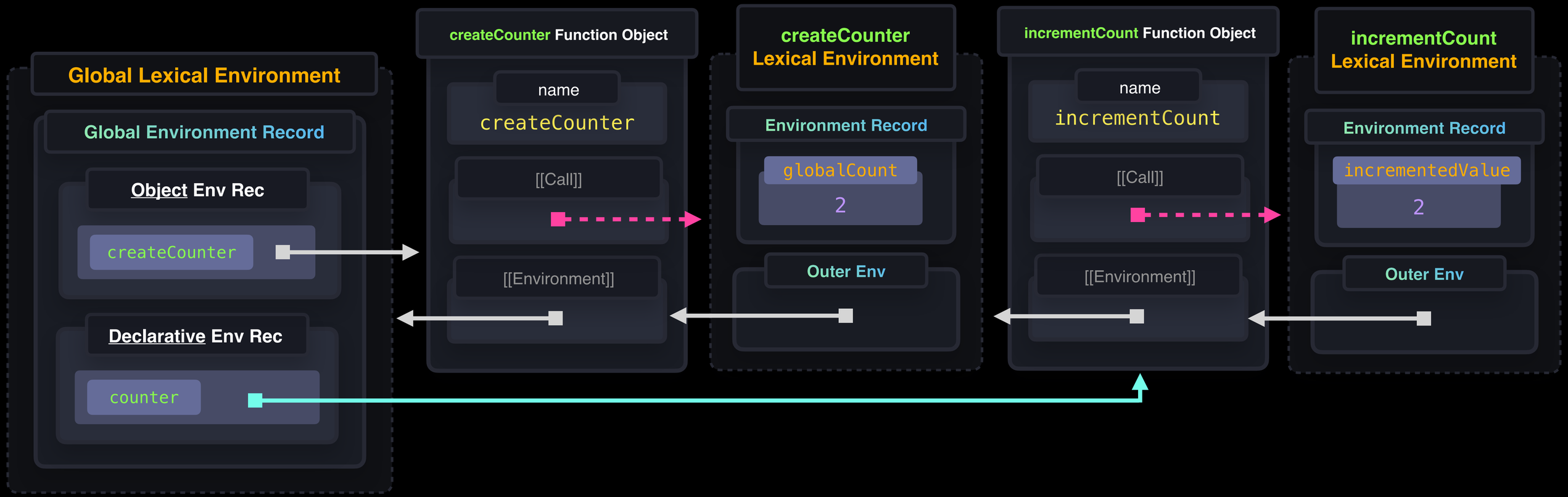
```
1  function createCounter() {
2    let globalCount = 0;
3
4    function incrementCount() {
5        let incrementedValue = ++globalCount;
6        return incrementedValue;
7      }
8
9    return { incrementCount }
10 };
11 const counter = createCounter();
12 console.log(counter.incrementCount());
13 console.log(counter.incrementCount());
14 console.log(createCounter().incrementCount());
```



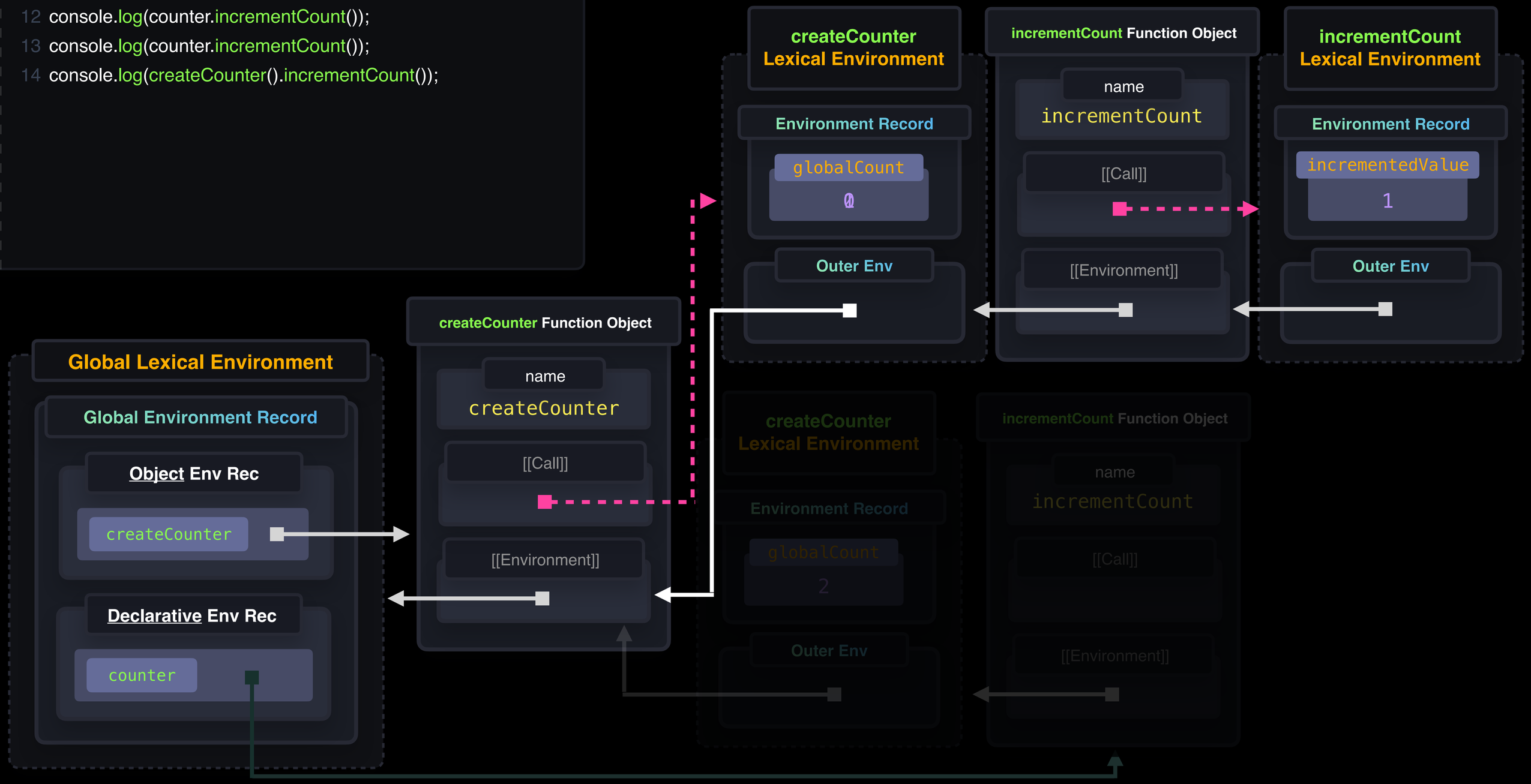
```
1  function createCounter() {
2    let globalCount = 0;
3
4    function incrementCount() {
5        let incrementedValue = ++globalCount;
6        return incrementedValue;
7      }
8
9    return { incrementCount }
10 };
11 const counter = createCounter();
12 console.log(counter.incrementCount());
13 console.log(counter.incrementCount());
14 console.log(createCounter().incrementCount());
```



```
1  function createCounter() {
2    let globalCount = 0;
3
4    function incrementCount() {
5        let incrementedValue = ++globalCount;
6        return incrementedValue;
7      }
8
9    return { incrementCount }
10 };
11 const counter = createCounter();
12 console.log(counter.incrementCount());
13 console.log(counter.incrementCount());
14 console.log(createCounter().incrementCount());
```



```
11 const counter = createCounter();
12 console.log(counter.incrementCount());
13 console.log(counter.incrementCount());
14 console.log(createCounter().incrementCount());
```



Question 7

Is the comparison true or false?

```
function createUserManager() {
  let user = null;

  return function(name) {
   "use strict"
      user = { name, createdAt: Date.now() };
      return user;
    }
};

const createUser = createUserManager();
createUser("John Doe") === createUser("John Doe");
```

A True

B False

Question 7

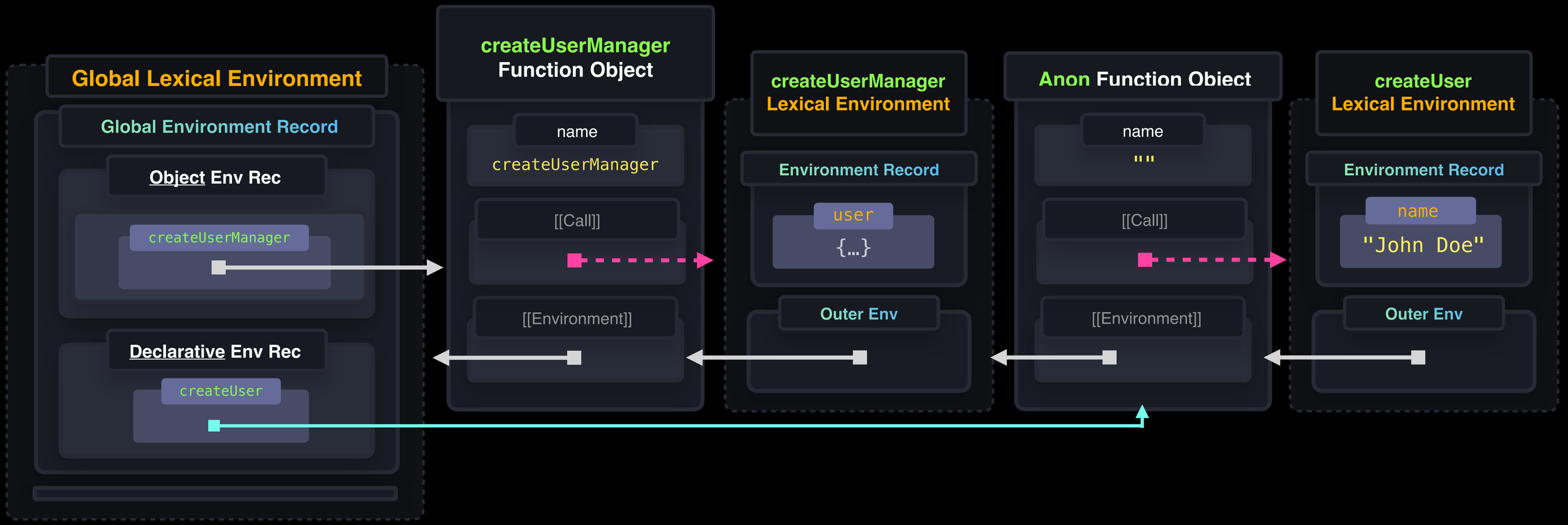
Is the comparison true or false?

```
function createUserManager() {
  let user = null;

  return function(name) {
   "use strict"
      user = { name, createdAt: Date.now() };
      return user;
    }
};

const createUser = createUserManager();
createUser("John Doe") === createUser("John Doe");
```

A True

B False

```
7        return user;
8      }
9    };
10
11   const createUser = createUserManager();
12   createUser("John Doe") === createUser("John Doe");
```



Question 8

What gets logged?

```
function createCounter(initialCount) {
  let count = initialCount;

  return function() {
   "use strict";
      count += 1;
      return count;
    }
};

const counter = createCounter(10);
counter();
counter();
console.log(counter());
```

A 1

B 11

C 13

D NaN

E ReferenceError

Question 8

What gets logged?

```
function createCounter(initialCount) {
  let count = initialCount;

  return function() {
   "use strict";
      count += 1;
      return count;
    }
};

const counter = createCounter(10);
counter();
counter();
console.log(counter());
```

A 1

B 11

C 13

D NaN

E ReferenceError

```
function createCounter(initialCount) {
  let count = initialCount;

  return function() {
    "use strict";
      count += 1;
      return count;
    }
};

const counter = createCounter(10);
counter();
counter();
console.log(counter());
```

Question 9

Which statements are correct?

A Hoisting is the process of moving functions and variables to the top of the file

B Variables declared with let and const are hoisted

C Variables declared with the var keyword are uninitialized

D Hoisting occurs during the execution phase

E import declarations are hoisted

Question 9

Which statements are correct?

A Hoisting is the process of moving functions and variables to the top of the file

B Variables declared with let and const are hoisted

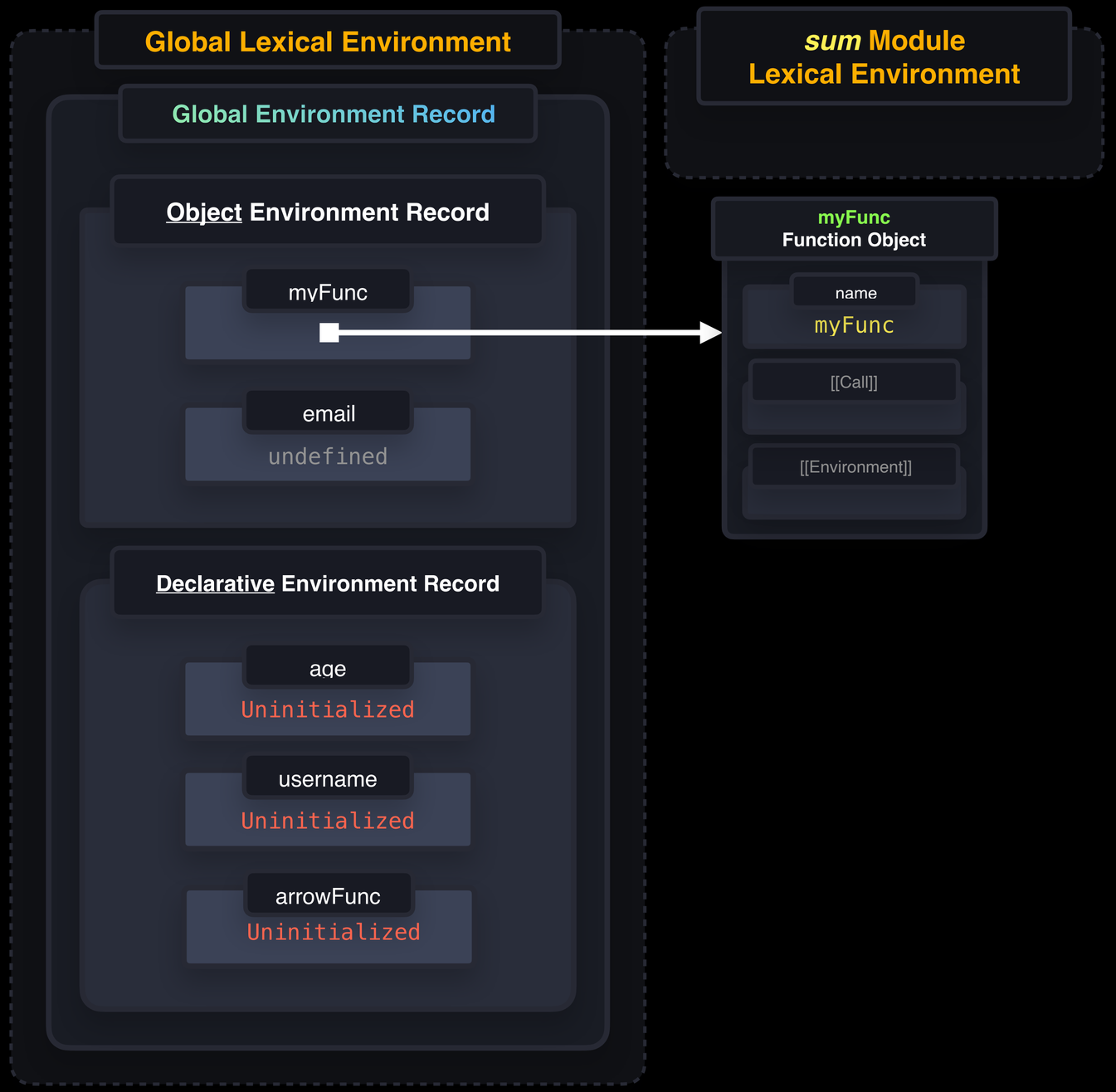
C Variables declared with the var keyword are uninitialized

D Hoisting occurs during the execution phase

E import declarations are hoisted

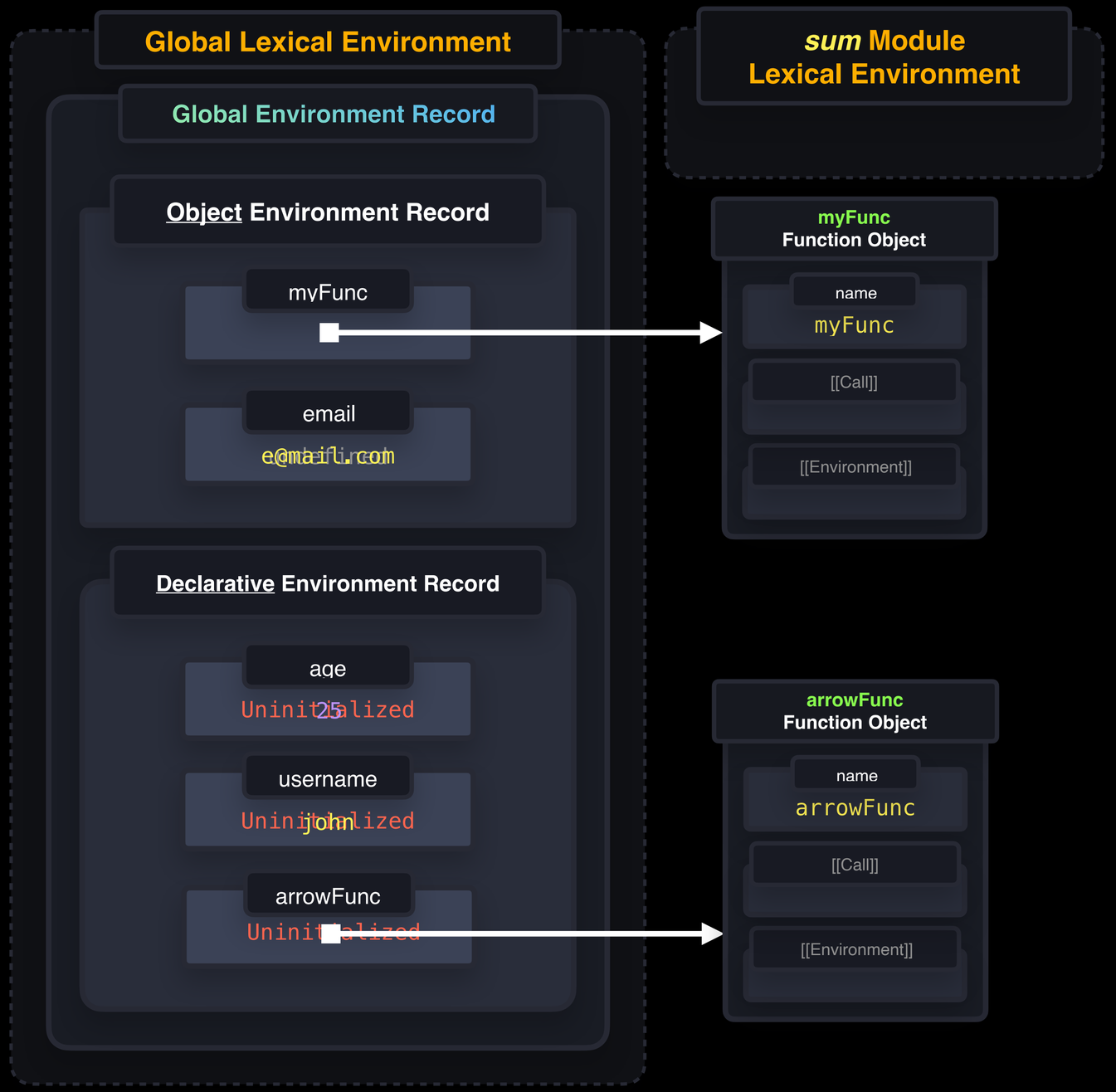
Creation Phase

```
1  import sum from "./sum";
2
3  const age = 25;
4  let username = "john";
5  var email = "e@mail.com";
6
7  function myFunc() {};
8  const arrowFunc = () => {};
```



Execution Phase

```
1  import sum from "./sum";
2
3  const age = 25;
4  let username = "john";
5  var email = "e@mail.com";
6
7  function myFunc() {};
8  const arrowFunc = () => {};
```



Question 10

What's the output if we run this code and orange is our favorite fruit?

```
const input = prompt("What fruit do you like?")
const css = "color: #FFFFFF;"

switch(input) {
  case "orange":
    const css = "color: #FFA500;"
    console.log("%cOrange!", css);
    break;
  case"lemon":
    const css = "color: #FFFF00;"
    console.log("%cYellow!", css);
    break;
  default:
    console.log("No color for you");
}
```

A Orange!

B %cOrange!

C No color for you

D SyntaxError

E Orange! (white text)

Question 10

What's the output if we run this code and orange is our favorite fruit?

```
const input = prompt("What fruit do you like?")
const css = "color: #FFFFFF;"

switch(input) {
  case "orange":
    const css = "color: #FFA500;"
    console.log("%cOrange!", css);
    break;
  case"lemon":
    const css = "color: #FFFF00;"
    console.log("%cYellow!", css);
    break;
  default:
    console.log("No color for you");
}
```

A Orange!

B %cOrange!

C No color for you

D SyntaxError

E Orange! (white text)

```
1  const input = prompt("What fruit do you like?”);
2  const css = "color: #FFFFFF;"
3
4  switch(input) {
5    case "orange":         {
6      const css = "color: #FFA500;"
7      console.log("%cOrange!", css);
8      break;
9      }
10     case "lemon":      {
11       const css = "color: #FFFF00;"
12       console.log("%cYellow!", css);
13       break;
14     }
15   default:
16     console.log("No color for you");
17 }
```

this Keyword

Global context

The value of the this keyword is the global object

```
1  console.log(this)        // window
```

Regular Functions

The value of the this keyword is the object on which the function is **invoked**.

```
function logThis() {
  console.log(this)
}

const obj = {
  logThis
}

logThis()       // window
obj.logThis()         // obj
```

Arrow Functions

The value of the this keyword in an arrow function is determined by the **lexical environment in which the arrow function was defined.**

```
1  const logThis = () => {
2    console.log(this)
3  }
4
5  const obj = {
6    logThis
7  }
8
9  logThis()        // window
10 obj.logThis()          // window
```

Classes

The value of the this keyword in constructor functions or classes is the **value of the newly created instance.**

```
1   class User {
2     getThis() {
3       return this
4     }
5   }
6
7   let user1 = new User();
8   let user2 = new User();
9
10  user1.getThis();          // user1
11  user2.getThis();          // user2
```

Strict mode

The value of the this keyword is undefined by default

```
1  "use strict"
2
3  function logThis() {
4    console.log(this)
5  }
6
7  const obj = {
8    logThis,
9    logThis2() {
10     return logThis()
11   }
12 }
13
14
15 logThis()       // undefined
16 obj.logThis()          // obj
17 obj.logThis2()         // undefined
```

Event Handlers

The value of the this keyword in event handlers using a regular function is the **element that received the event**

```
button.addEventListener(
  "click",
  function() {
    console.log(this)
  }
)

button.click();
// <button>…</button>
```

.call()

Calls a function with a given this value

.bind()

Creates a new function with a specified this value and optional initial arguments.

.apply()

Calls a function with a given this value and arguments provided as an array

```
1   function greet(text = "Welcome") {
2     console.log(`${text}, ${this.username}`)
3   }
4
5   greet.call({ username: "John" });
6   // Welcome, John
7
8   greet.bind({ username: "John" })();
9   // Welcome, John
10
11  greet.apply(
12      { username: "John" },
13      ["Hi there"]
14  )
15  // Hi there, John
```

Question 11

What gets logged?

```
function logThis() {
  console.log(this)
}

const obj = {
  logThis,
  logThis2() {
    return logThis()
  },
  logThis3() {
    return obj.logThis();
  }
}

obj.logThis();
obj.logThis2();
obj.logThis3();
```

A	obj	window	obj
B	obj	obj	obj
C	window	obj	obj
D	window	window	obj
E	window	window	window

Question 11

What gets logged?

```
function logThis() {
  console.log(this)
}

const obj = {
  logThis,
  logThis2() {
    return logThis()
  },
  logThis3() {
    return obj.logThis();
  }
}

obj.logThis();
obj.logThis2();
obj.logThis3();
```

A	obj	window	obj
B	obj	obj	obj
C	window	obj	obj
D	window	window	obj
E	window	window	window

```
1   function logThis() {
2     console.log(this)
3   }
4
5   const obj = {
6     logThis,
7     logThis2() {
8       return logThis()
9     },
10    logThis3() {
11      return obj.logThis()
12    }
13  }
14
15  obj.logThis();                    // obj
16  obj.logThis2();                   // window
17  obj.logThis3();                   // obj
```

```
1   function logThis() {
2     console.log(this)
3   }
4
5   const obj = {
6     logThis,
7     logThis2() {
8       return logThis()
9     },
10    logThis3() {
11      return obj.logThis()
12    }
13  }
14
15  obj.logThis();                    // obj
16  obj.logThis2();                   // window
17  obj.logThis3();                   // obj
```

```
1   function logThis() {
2     console.log(this)
3   }
4
5   const obj = {
6     logThis,
7     logThis2() {
8       return logThis()
9     },
10    logThis3() {
11      return obj.logThis()
12    }
13  }
14
15  obj.logThis();                    // obj
16  obj.logThis2();                     // window
17  obj.logThis3();                   // obj
```

Question 12

What will the this keyword refer to for each invocation

```
1   const objA = {
2       foo() {
3         console.log(this)
4       },
5       bar: () => console.log(this),
6   }
7
8   const objB  = {
9     foo: objA.foo,
10    bar: () => objA.bar(),
11    baz() { objA.foo() }
12  }
13
14  objB.foo();
15  objB.bar();
16  objB.baz();
```

A objA objA objA

B objB objB objB

C objA window objA

D objB window objB

E objB window objA

Question 12

What will the this keyword refer to for each invocation

```
1   const objA = {
2       foo() {
3         console.log(this)
4       },
5       bar: () => console.log(this),
6   }
7
8   const objB  = {
9     foo: objA.foo,
10    bar: () => objA.bar(),
11    baz() { objA.foo() }
12  }
13
14  objB.foo();
15  objB.bar();
16  objB.baz();
```

A objA objA objA

B objB objB objB

C objA window objA

D objB window objB

E objB window objA

```
1   const objA = {
2       foo() {
3         console.log(this)
4       },
5       bar: () => console.log(this),
6   }
7
8   const objB  = {
9     foo: objA.foo,
10    bar: () => objA.bar(),
11    baz() { objA.foo() }
12  }
13
14  objB.foo();          // objB
15  objB.bar();          // window
16  objB.baz();          // objA
```

```
1   const objA = {
2       foo() {
3         console.log(this)
4       },
5       bar: () => console.log(this),
6   }
7
8   const objB  = {
9     foo: objA.foo,
10    bar: () => objA.bar(),
11    baz() { objA.foo() }
12  }
13
14  objB.foo();          // objB
15  objB.bar();          // window
16  objB.baz();          // objA
```

```
1   const objA = {
2       foo() {
3         console.log(this)
4       },
5       bar: () => console.log(this),
6   }
7
8   const objB  = {
9     foo: objA.foo,
10    bar: () => objA.bar(),
11    baz() { objA.foo() }
12  }
13
14  objB.foo();          // objB
15  objB.bar();          // window
16  objB.baz();          // objA
```

Question 13

What gets logged?

```
function logThis() {
  console.log(this)
}

const obj1 = {
  foo: logThis,
  bar: () => logThis(),
    baz() { logThis() },
    qux() { logThis.call(this) }
}

const obj2 = { ...obj1, foo2: () => obj1.foo() }

for (const key in obj2) {
    obj2[key]()
}
```

A	window	window	window	window	window
B	obj1	window	window	obj1	obj1
C	obj2	window	window	obj2	obj1
D	obj1	obj1	obj1	obj1	obj1
E	obj1	window	obj2	obj1	obj1

Question 13

What gets logged?

```
1  function logThis() {
2    console.log(this)
3  }
4
5  const obj1 = {
6    foo: logThis,
7    bar: () => logThis(),
8      baz() { logThis() },
9      qux() { logThis.call(this) }
10 }
11
12 const obj2 = { ...obj1, foo2: () => obj1.foo() }
13
14 for (const key in obj2) {
15     obj2[key]()
16 }
```

A	window	window	window	window	window
B	obj1	window	window	obj1	obj1
C	obj2	window	window	obj2	obj1
D	obj1	obj1	obj1	obj1	obj1
E	obj1	window	obj2	obj1	obj1

```
1   function logThis() {
2     console.log(this)
3   }
4
5   const obj1 = {
6     foo: logThis,
7     bar: () => logThis(),
8       baz() { logThis() },
9       qux() { logThis.call(this) }
10  }
11
12  const obj2 = {
13    foo: logThis,
14    bar: () => logThis(),
15      baz() { logThis() },
16      qux() { logThis.call(this) },
17      foo2: () => obj1.foo()
18  }
19
20  for (const key in obj2) {
21      obj2[key]()
22  }
```

C	obj2	window	window	obj2	obj1

Question 14

What gets logged?

```
function logThis() {
  console.log(this)
}

const obj = {
  logThis,
  logThisInArrow: () => console.log(this),
  logThisNested() {
    const nestedFunc = () => console.log(this);
    nestedFunc()
  },
}

obj.logThis();
obj.logThisInArrow();
obj.logThisNested();
```

A	obj	window	obj
B	obj	obj	obj
C	window	window	window
D	obj	window	window
E	window	obj	window

Question 14

What gets logged?

```
function logThis() {
  console.log(this)
}

const obj = {
  logThis,
  logThisInArrow: () => console.log(this),
  logThisNested() {
    const nestedFunc = () => console.log(this);
    nestedFunc()
  },
}

obj.logThis();
obj.logThisInArrow();
obj.logThisNested();
```

A	obj	window	obj
B	obj	obj	obj
C	window	window	window
D	obj	window	window
E	window	obj	window

```
function logThis() {
  console.log(this)
}

const obj = {
  logThis,
  logThisInArrow: () => console.log(this),
  logThisNested() {
    const nestedFunc = () => console.log(this);
    nestedFunc()
  },
}

obj.logThis();                    //obj
obj.logThisInArrow();             // window
obj.logThisNested();              // obj
```

Question 15

What gets logged?

```
const obj = {
  logThis() {
    console.log(this)
  },
  logThis2() {
    function logThisInner() {
        console.log(this)
      }
      return logThisInner.apply(this)
  },
};

const { logThis, logThis2 } = obj;

logThis();
logThis2();
obj.logThis();
obj.logThis2();
```

A	window	window	window	window
B	window	obj	window	obj
C	obj	window	obj	window
D	obj	obj	obj	obj
E	window	window	obj	obj

Question 15

What gets logged?

```
1  const obj = {
2    logThis() {
3      console.log(this)
4    },
5    logThis2() {
6      function logThisInner() {
7          console.log(this)
8        }
9        return logThisInner.apply(this)
10   },
11 };
12
13 const { logThis, logThis2 } = obj;
14
15 logThis();
16 logThis2();
17 obj.logThis();
18 obj.logThis2();
```

A	window	window	window	window
B	window	obj	window	obj
C	obj	window	obj	window
D	obj	obj	obj	obj
E	window	window	obj	obj

```
1   const obj = {
2     logThis() {
3       console.log(this)
4     },
5     logThis2() {
6       function logThisInner() {
7           console.log(this)
8         }
9         return logThisInner.apply(this)
10    },
11  };
12
13  const { logThis, logThis2 } = obj;
14
15  logThis();          // window
16  logThis2();           // window
17  obj.logThis();            // obj
18  obj.logThis2();             // obj
```

Classes and Prototypes

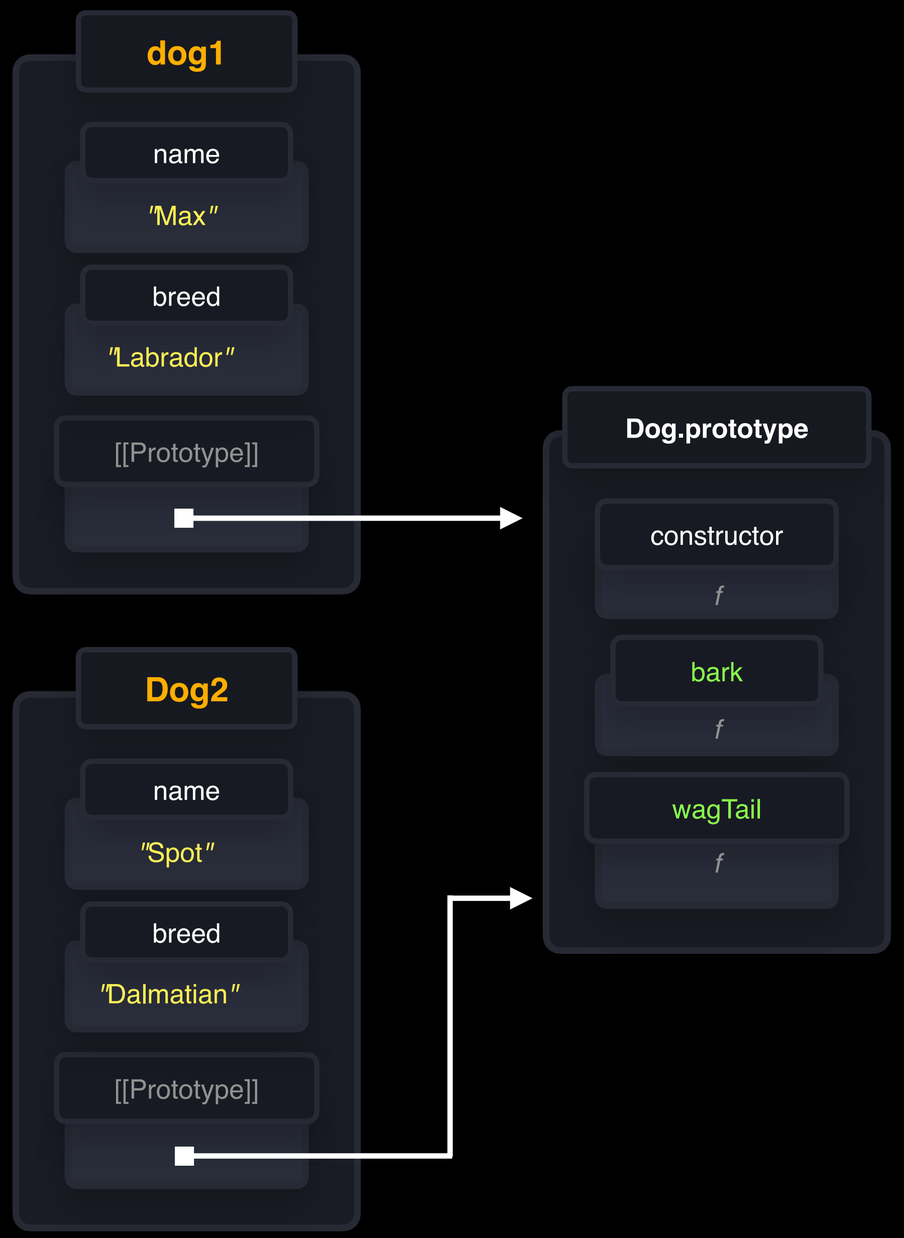
```
class Animal {
    constructor(name) {
        this.name = name;
    }
}
```

```
class Mammal extends Animal {
    constructor(name) {
      super(name);
    }

    breathe() { … }
}
```

```
class Canine extends Mammal {
    constructor(name) {
      super(name);
    }

    howl() { … }
}
```

```
class Dog extends Canine {
    constructor(name) {
      super(name);
    }

    wagTail() { … }
}
```

```
1  class Dog {
2      constructor(name, breed) {
3          this.name = name;
4          this.breed = breed;
5      }
6
7      bark() { }
8      wagTail() { }
9  }
10
11 const dog1 = new Dog("Max", "Labrador");
12 const dog2 = new Dog("Spot", "Dalmatian");
```



```
dog1.__proto__ (deprecated)
```

```
Object.getPrototypeOf(dog1)
```

Question 16

Creating a new User instance would create a new login function in memory each time

```
class User {
  constructor(username) {
    this.username = username;
  }

  login() { … }
}

const user1 = new User("johndoe");
const user2 = new User("janedoe");
```

A True

B False

Question 16

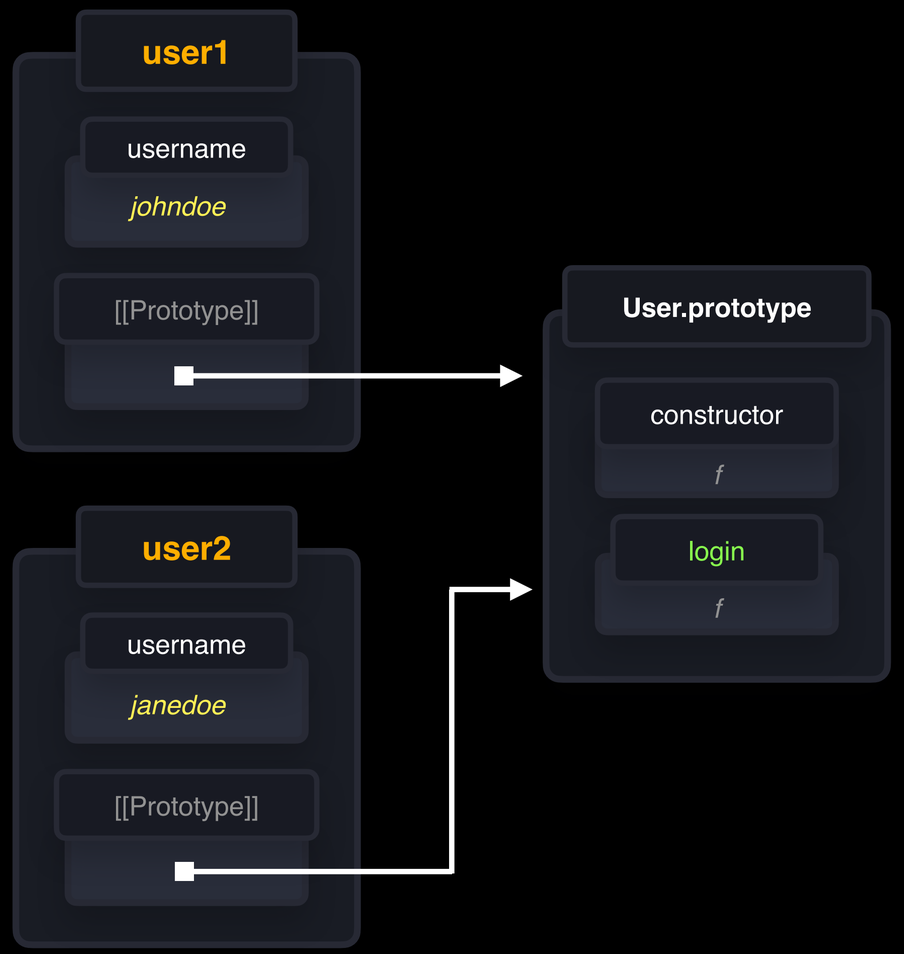
Creating a new User instance would create a new login function in memory each time

```
class User {
  constructor(username) {
    this.username = username;
  }

  login() { … }
}

const user1 = new User("johndoe");
const user2 = new User("janedoe");
```

A True

B False

```
1   class User {
2     constructor(username) {
3       this.username = username;
4     }
5
6     login() { … }
7   }
8
9   const user1 = new User("johndoe");
10  const user2 = new User("janedoe");
```



Question 17

Which of the following statements are true?

```
1  class Dog {
2    constructor(name) {
3      this.username = username;
4      this.wagTail = () => {
5        return "Wagging tail!"
6      }
7    }
8
9    bark() {
10     return "Woof!"
11   }
12 }
13
14 const dog1 = new Dog("Max");
15 const dog2 = new Dog("Spot");
```

A dog1.wagTail() === dog2.wagTail()

B dog1.wagTail === dog2.wagTail

C dog1.bark === dog2.bark

D Object.getPrototypeOf(dog1) ===
Object.getPrototypeOf(dog2)

E dog1.constructor === dog2.constructor

Question 17

Which of the following statements are true?

```
1  class Dog {
2      constructor(name) {
3          this.username = username;
4          this.wagTail = () => {
5            return "Wagging tail!"
6          }
7      }
8
9      bark() {
10       return "Woof!"
11     }
12 }
13
14 const dog1 = new Dog("Max");
15 const dog2 = new Dog("Spot");
```

A dog1.wagTail() === dog2.wagTail()

B dog1.wagTail === dog2.wagTail

C dog1.bark === dog2.bark

D Object.getPrototypeOf(dog1) ===
Object.getPrototypeOf(dog2)

E dog1.constructor === dog2.constructor

```
1   class Dog {
2       constructor(name) {
3           this.username = username;
4           this.wagTail = () => {
5             return "Wagging tail!"
6           }
7       }
8
9       bark() {
10        return "Woof!"
11      }
12  }
13
14  const dog1 = new Dog("Max");
15  const dog2 = new Dog("Spot");
```

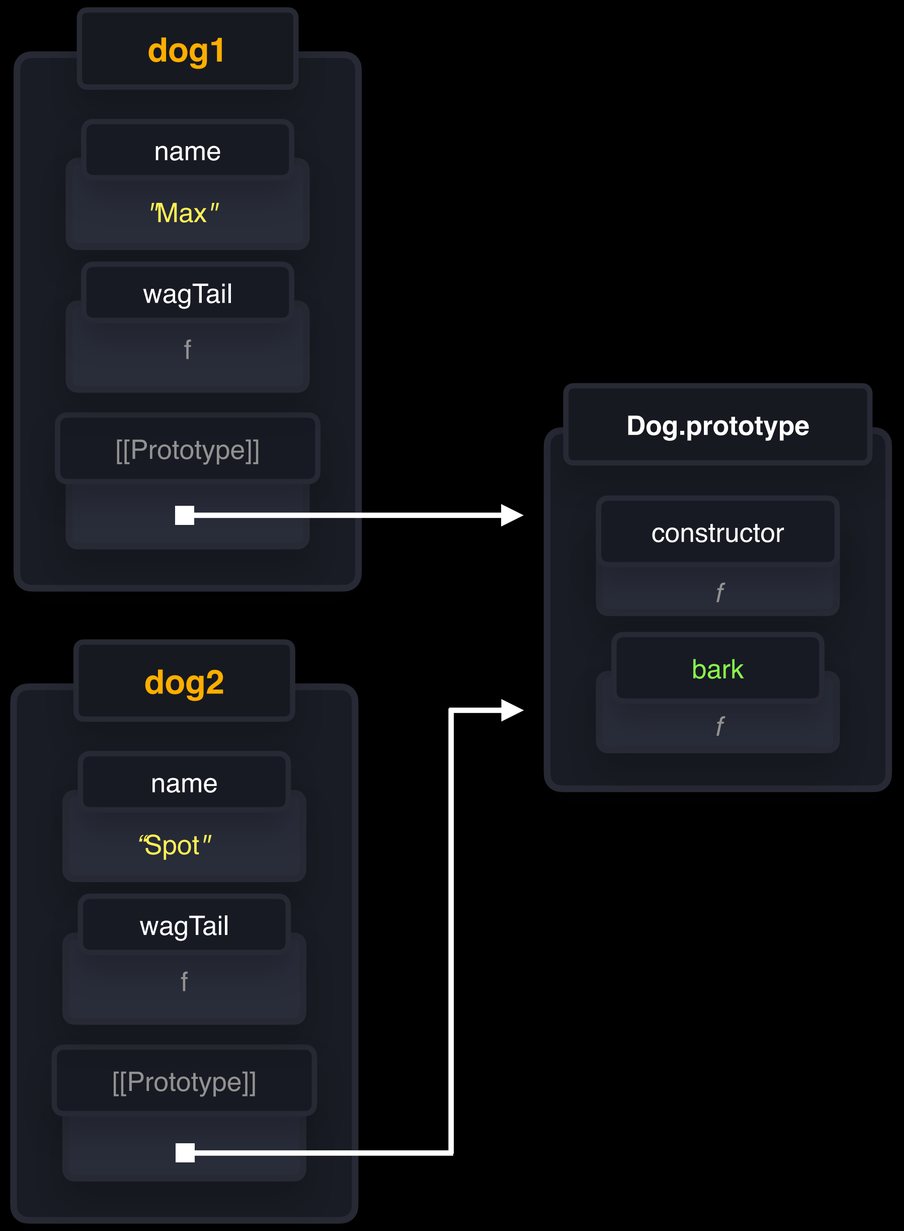
A dog1.wagTail() === dog2.wagTail()

B dog1.wagTail === dog2.wagTail

C dog1.bark === dog2.bark

D Object.getPrototypeOf(dog1) ===
Object.getPrototypeOf(dog2)

E dog1.constructor === dog2.constructor



Question 18

What gets logged?

```
class Counter {
    constructor(initialCount = 0) {
        this.count = initialCount;
    }

    increment() { return this.count++ }
}

const counterOne = new Counter(1);
counterOne.increment();
const counterTwo = Object.create(counterOne);
counterTwo.increment();

console.log(
  counterOne.count,
  counterTwo.count
)
```

A	2 2
B	2 3
C	1 1
D	3 2
E	TypeError

Question 18

What gets logged?

```
1  class Counter {
2      constructor(initialCount = 0) {
3          this.count = initialCount;
4      }
5
6      increment() { return this.count++ }
7  }
8
9  const counterOne = new Counter(1);
10 counterOne.increment();
11 const counterTwo = Object.create(counterOne);
12 counterTwo.increment();
13
14 console.log(
15   counterOne.count,
16   counterTwo.count
17 )
```

A 2 2

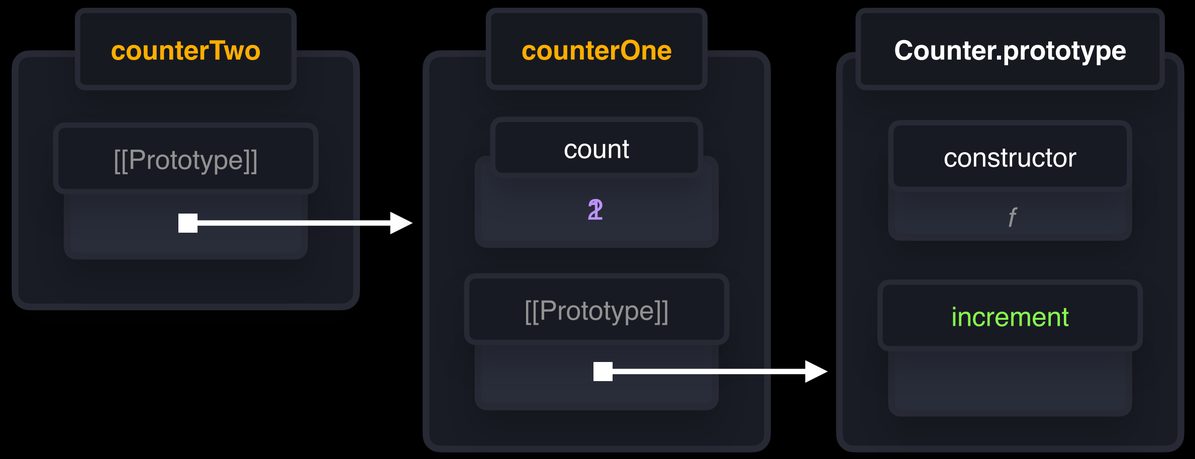
B 2 3

C 1 1

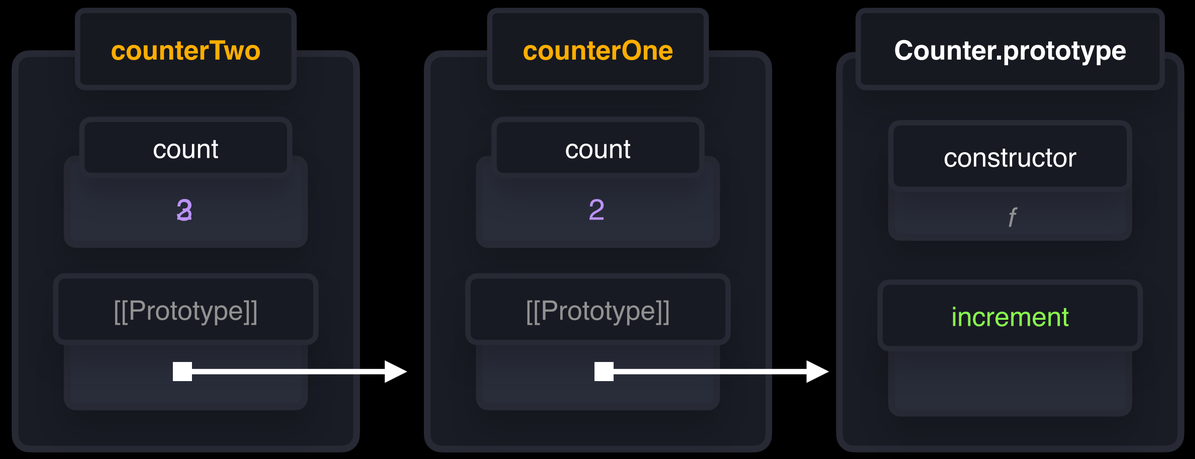
D 3 2

E TypeError

```
1   class Counter {
2       constructor(initialCount = 0) {
3           this.count = initialCount;
4       }
5
6       increment() { return this.count++ }
7   }
8
9   const counterOne = new Counter(1);
10  counterOne.increment();
11  const counterTwo = Object.create(counterOne);
12  counterTwo.increment();
```



```
1   class Counter {
2       constructor(initialCount = 0) {
3           this.count = initialCount;
4       }
5
6       increment() { return this.count++ }
7   }
8
9   const counterOne = new Counter(1);
10  counterOne.increment();
11  const counterTwo = Object.create(counterOne);
12  counterTwo.increment();
```



Question 19

What gets logged?

```
class Chameleon {
  constructor(color = “green”) {
    this.color = color;
  }

  static changeColor(newColor) {
    this.color = newColor;
    return this.color;
  }
}

const freddie = new Chameleon("green");
freddie.changeColor("orange");
```

A orange

B purple

C green

D TypeError

E *undefined*

Question 19

What gets logged?

```
class Chameleon {
  constructor(color = “green”) {
    this.color = color;
  }

  static changeColor(newColor) {
    this.color = newColor;
    return this.color;
  }
}

const freddie = new Chameleon("green");
freddie.changeColor("orange");
```

A orange

B purple

C green

D TypeError

E *undefined*

```
1   class Chameleon {
2       constructor(color = “green”) {
3           this.color = color;
4       }
5
6       static changeColor(newColor) {
7         this.color = newColor;
8           return this.color;
9     }
10  }
11
12  Chameleon.changeColor // function
13  new Chameleon().changeColor // undefined
```

Question 20

Which statements are true?

```
1  class User {
2      constructor(username) {
3          this.username = username;
4      }
5
6      login() { … }
7  }
8
9  const user = new User("johndoe");
```

A Object.getPrototypeOf(user) === User.prototype

B Object.getPrototypeOf(user) === Object.getPrototypeOf(User)

C user.prototype === User.prototype

D Object.getPrototypeOf(user) === User.constructor

Question 20

Which statements are true?

```
1  class User {
2      constructor(username) {
3          this.username = username;
4      }
5
6      login() { … }
7  }
8
9  const user = new User("johndoe");
```

A Object.getPrototypeOf(user) === User.prototype

B Object.getPrototypeOf(user) === Object.getPrototypeOf(User)

C user.prototype === User.prototype

D Object.getPrototypeOf(user) === User.constructor

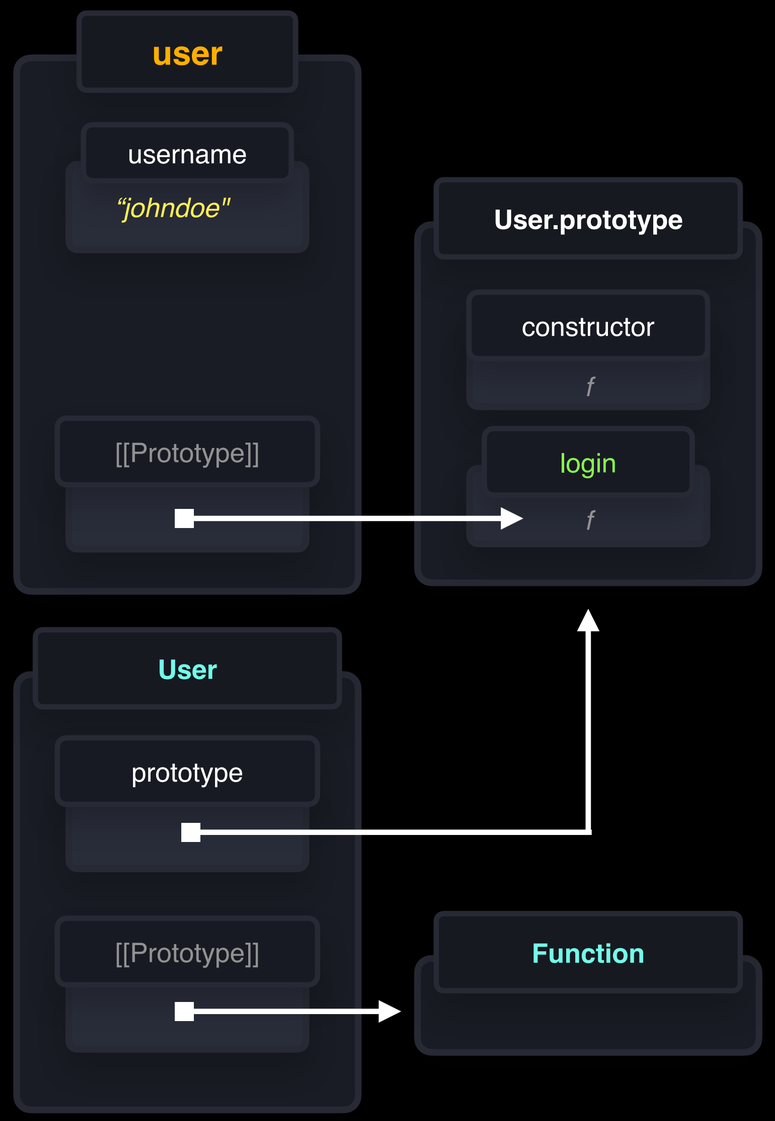
```
1  class User {
2      constructor(username) {
3          this.username = username;
4      }
5
6      login() { … }
7  }
8
9  const user = new User("johndoe");
```

A Object.getPrototypeOf(user) === User.prototype

B Object.getPrototypeOf(user) === Object.getPrototypeOf(User)

C user.prototype === User.prototype

D Object.getPrototypeOf(user) === User.constructor



Generators & Iterators

```
1   function* genFunc() {
2     yield 1;
3     yield 2;
4     yield 3;
5     return 4;
6   }
7
8   const iterator = genFunc();
9
10  iterator.next();            // { value: 1, done: false }
11
12
13
```

iterator

state	Running
next	function
return	function
throw	function

```
function* genFunc() {
  yield 1;
  yield 2;
  yield 3;
  return 4;
}

const iterator = genFunc();

iterator.next();          // { value: 1, done: false }
iterator.next();          // { value: 2, done: false }
```

iterator	
state	Running
next	function
return	function
throw	function

```
1   function* genFunc() {
2     yield 1;
3     yield 2;
4     yield 3;
5     return 4;
6   }
7
8   const iterator = genFunc();
9
10  iterator.next();          // { value: 1, done: false }
11  iterator.next();          // { value: 2, done: false }
12  iterator.next();          // { value: 3, done: false }
13  iterator.next();          // { value: 4, done: true }
```

iterator

state	Closed
next	function
return	function
throw	function

```
1  function* generatorFunction() {
2    yield 1;
3    yield 2;
4    yield 3;
5    return 4;
6  }
7
8
9  console.log([...generatorFunction()]); // [1, 2, 3]
10
11 for (const value of generatorFunction()) {
12     console.log(value)
13 } // 1  2  3
```

Question 21

What gets logged?

```
function* count() {
  yield 1;
  yield 2;
  return 3;
}

for (const value of count()) {
    console.log(value)
}
```

A 1 2 3

B 1 2 Promise

C 3

D 1

E 1 2

Question 21

What gets logged?

```
function* count() {
  yield 1;
  yield 2;
  return 3;
}

for (const value of count()) {
    console.log(value)
}
```

A 1 2 3

B 1 2 Promise

C 3

D 1

E 1 2

```
1  function* count() {
2    yield 1;
3    yield 2;
4    return 3;
5  }
6
7  for (const value of count()) {
8      console.log(value)
9  }
```

Question 22

To create a custom iterable object in JavaScript, the object must implement the `Symbol.iterator` method that returns an iterator.

The iterator must implement what method?

A next

B iterator

C iterate

D forEach

E None

Question 22

To create a custom iterable object in JavaScript, the object must implement the `Symbol.iterator` method that returns an iterator.

The iterator must implement what method?

A next

B iterator

C iterate

D forEach

E None

```
1  const range = {
2    from: 1,
3    to: 5,
4    [Symbol.iterator]() {
5      current: this.from,
6      last: this.to,
7      return {
8        next() {
9          if (this.current <= this.last) {
10           return { done: false, value: this.current++ }
11         } else {
12           return { done: true }
13         }
14       }
15     }
16   }
17 }
18
19 console.log([...range]) // [1, 2, 3, 4, 5]
```

```
1  const range = {
2    from: 1,
3    to: 5,
4    *[Symbol.iterator]() {
5      for (let i = this.from; i <= this.to; I++) {
6        yield value;
7      }
8    }
9  }
10
11 console.log([...range]) // [1, 2, 3, 4, 5]
```

Question 23

What gets logged?

```
1  async function* range(start, end) {
2    for (let i = start; i <= end; i++) {
3      yield Promise.resolve(i);
4    }
5  }
6
7  (async () => {
8    const gen = range(1, 3);
9    for await (const item of gen) {
10     console.log(item)
11   }
12 })();
```

A Promise{1} Promise{2} Promise{3}

B 3x Promise{<pending>}

C 1 2 3

D 3x undefined

E [1, 2, 3] undefined undefined

Question 23

What gets logged?

```
1  async function* range(start, end) {
2    for (let i = start; i <= end; i++) {
3      yield Promise.resolve(i);
4    }
5  }
6
7  (async () => {
8    const gen = range(1, 3);
9    for await (const item of gen) {
10     console.log(item)
11   }
12 })();
```

A Promise{1} Promise{2} Promise{3}

B 3x Promise{<pending>}

C 1 2 3

D 3x undefined

E [1, 2, 3] undefined undefined

```javascript
1  async function* range(start, end) {
2    for (let i = start; i <= end; i++) {
3      yield Promise.resolve(i);
4      }
5  }
6
7  (async () => {
8    const gen = range(1, 3);
9    for await (const item of gen) {
10     console.log(item)
11   }
12 })();
```

Simplified but same idea

```javascript
1  async function* range() {
2    yield Promise.resolve(1);
3    yield Promise.resolve(2);
4    yield Promise.resolve(3);
5  }
6
7  (async () => {
8    for await (const item of range()) {
9      console.log(item)
10   }
11 })();
```

Question 24

What gets logged?

```
function* gen1() {
  yield 2;
  yield 3;
}

function* gen2() {
  yield 1;
  yield* gen1();
  yield 4;
}

console.log([...gen2()])
```

A [1, 2, 3, 4]

B [1, [2, 3], 4]

C SyntaxError

D [1, 4, 2, 3]

Question 24

What gets logged?

```
function* gen1() {
  yield 2;
  yield 3;
}

function* gen2() {
  yield 1;
  yield* gen1();
  yield 4;
}

console.log([...gen2()])
```

A [1, 2, 3, 4]

B [1, [2, 3], 4]

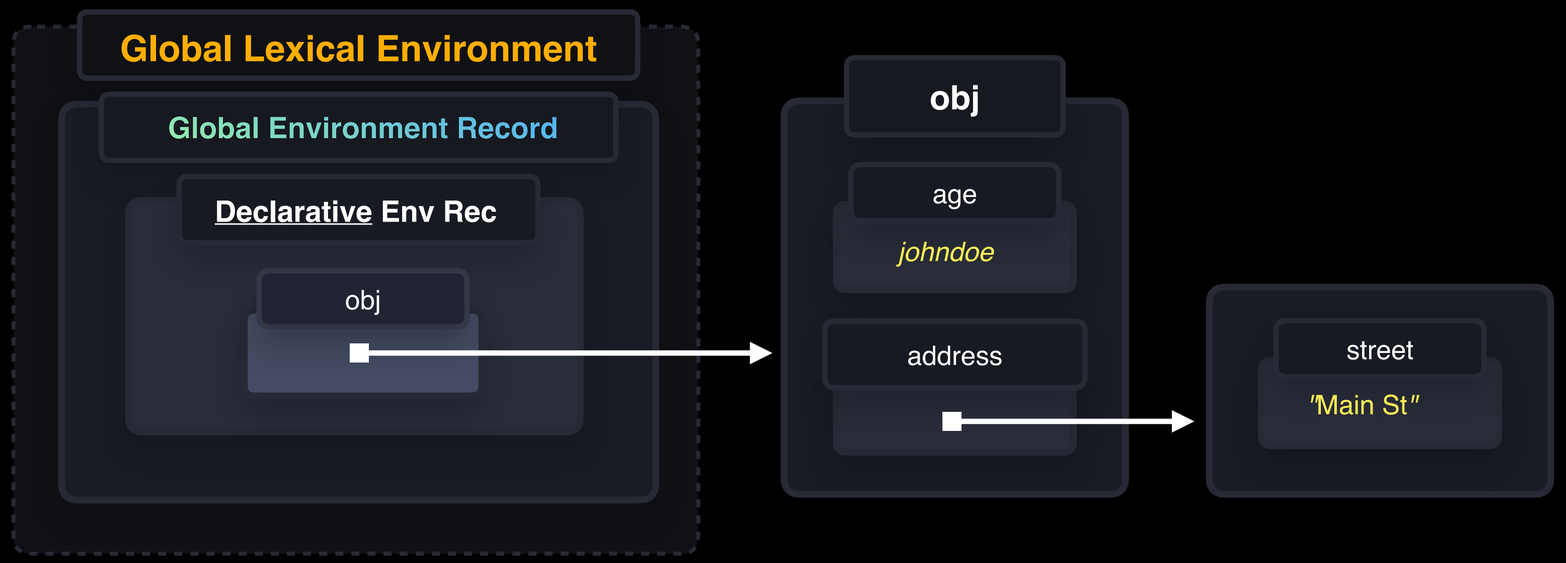
C SyntaxError

D [1, 4, 2, 3]

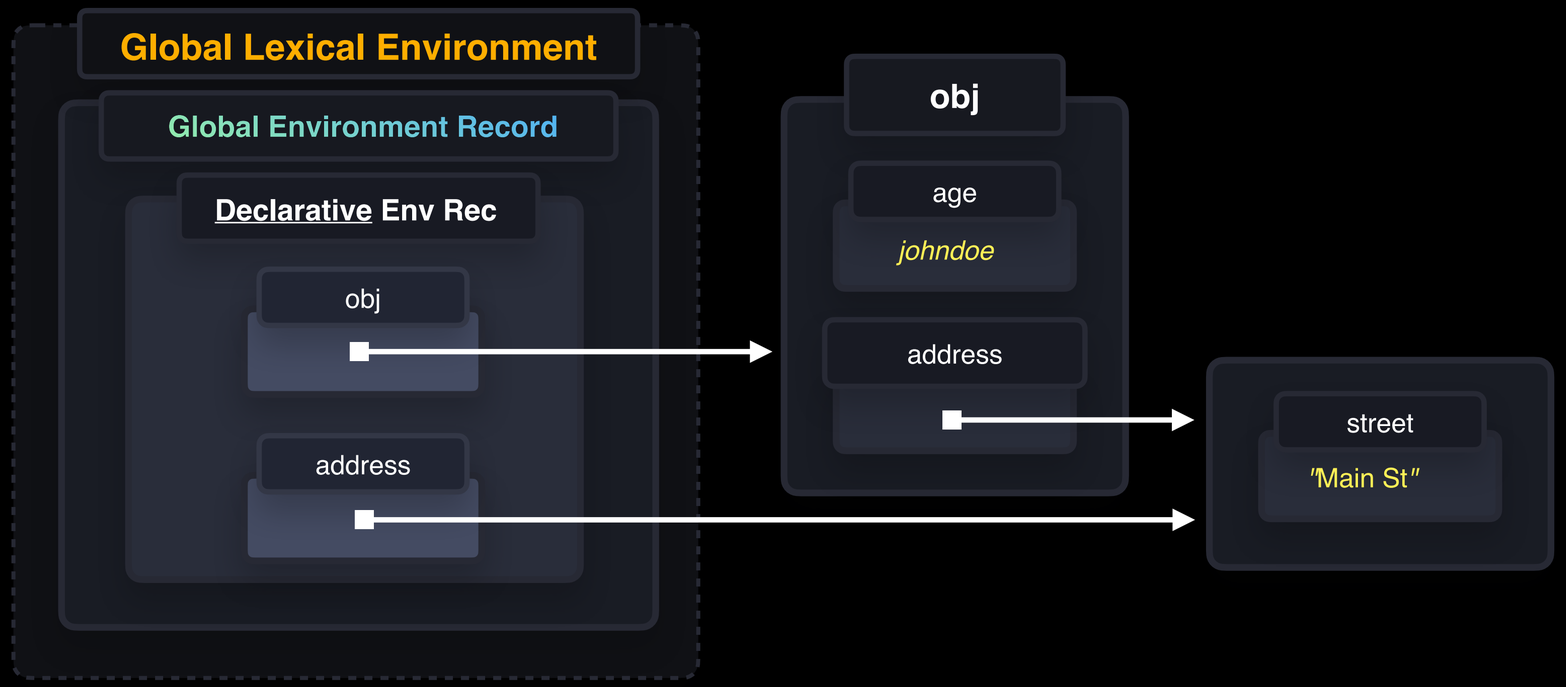
```
1   function* gen1() {
2     yield 2;
3     yield 3;
4   }
5
6   function* gen2() {
7     yield 1;
8     yield* gen1();
9     yield 4;
10  }
11
12  console.log([…gen2()])
```

Garbage Collection

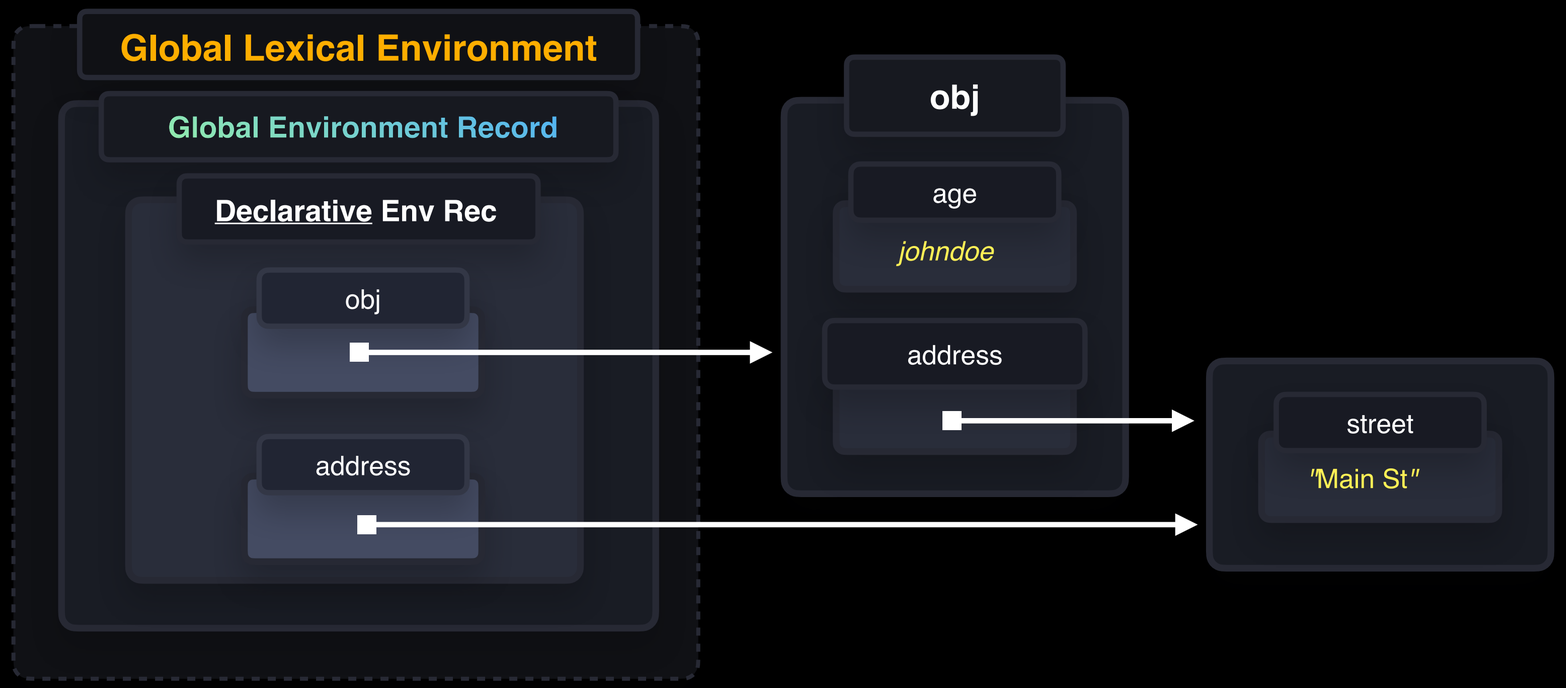
```
1  const obj = {
2    age: 25,
3    address: {
4      street: "Main St"
5    }
6  }
```



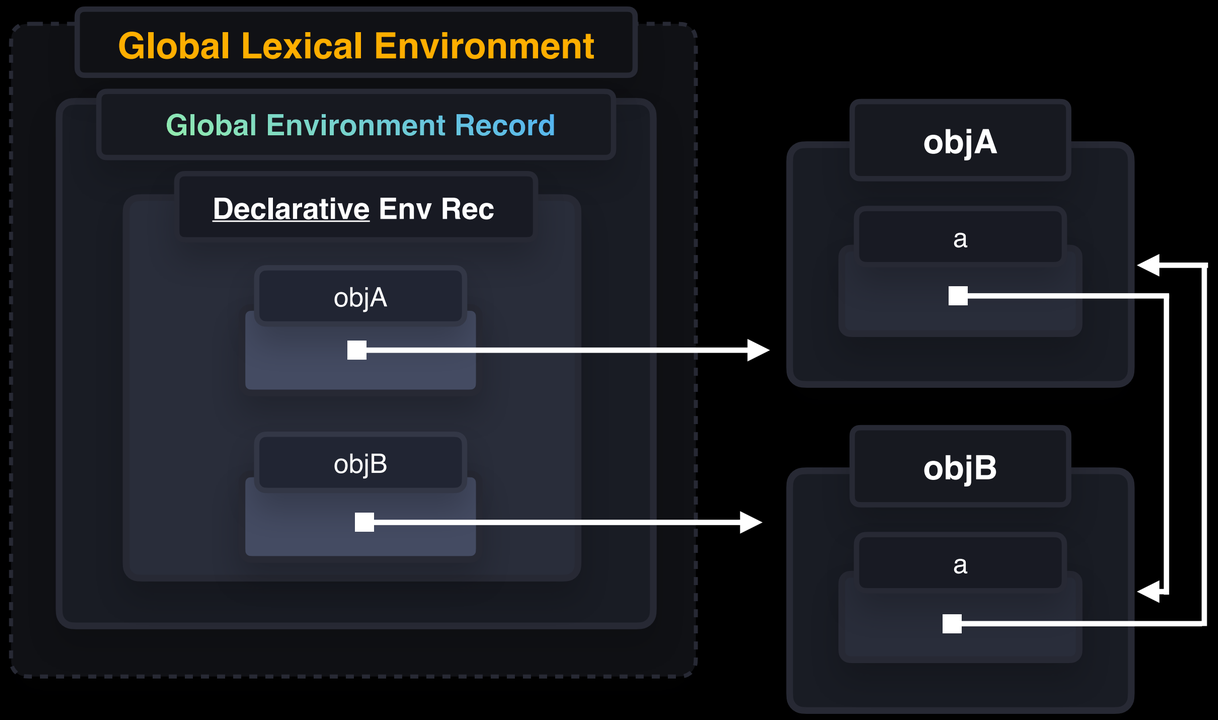
```
1  const obj = {
2    age: 25,
3    address: {
4      street: "Main St"
5    }
6  }
7
8  const address = obj.address;
```



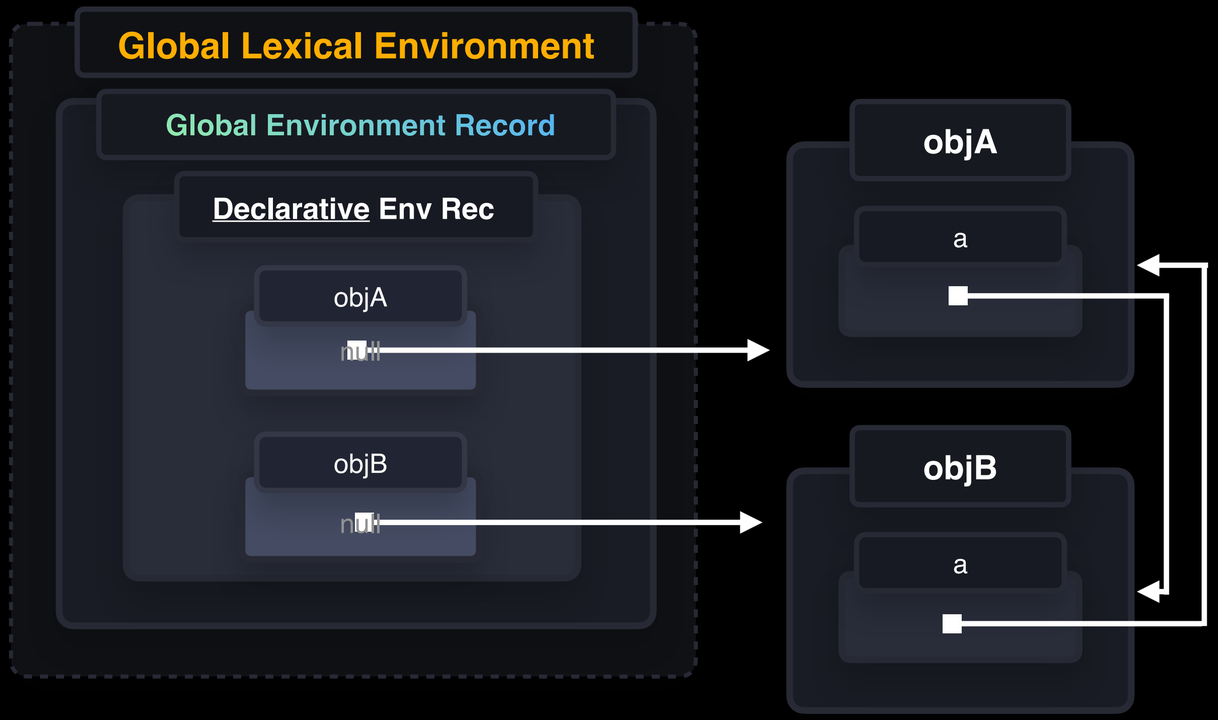
```
1   const obj = {
2     age: 25,
3     address: {
4       street: "Main St"
5     }
6   }
7
8   const address = obj.address;
9   obj = null;
10
```



```
1  const objA = { };
2  const objB = { };
3
4  objA.a = objB;
5  objB.a = objA;
6
7
8
```



```
1  const objA = { };
2  const objB = { };
3
4  objA.a = objB;
5  objB.a = objA;
6
7  objA = null;
8  objB = null;
```



Question 25

Which of the following statements are true about a WeakMap?

A	Keys are strongly referenced, values are weakly referenced
B	Keys are weakly referenced, values are strongly referenced
C	It is enumerable, allowing iteration over its elements
D	It can have primitive data types as keys
E	We can use the keys() method on a WeakMap, but not values()

Question 25

Which of the following statements are true about a WeakMap?

A Keys are strongly referenced, values are weakly referenced

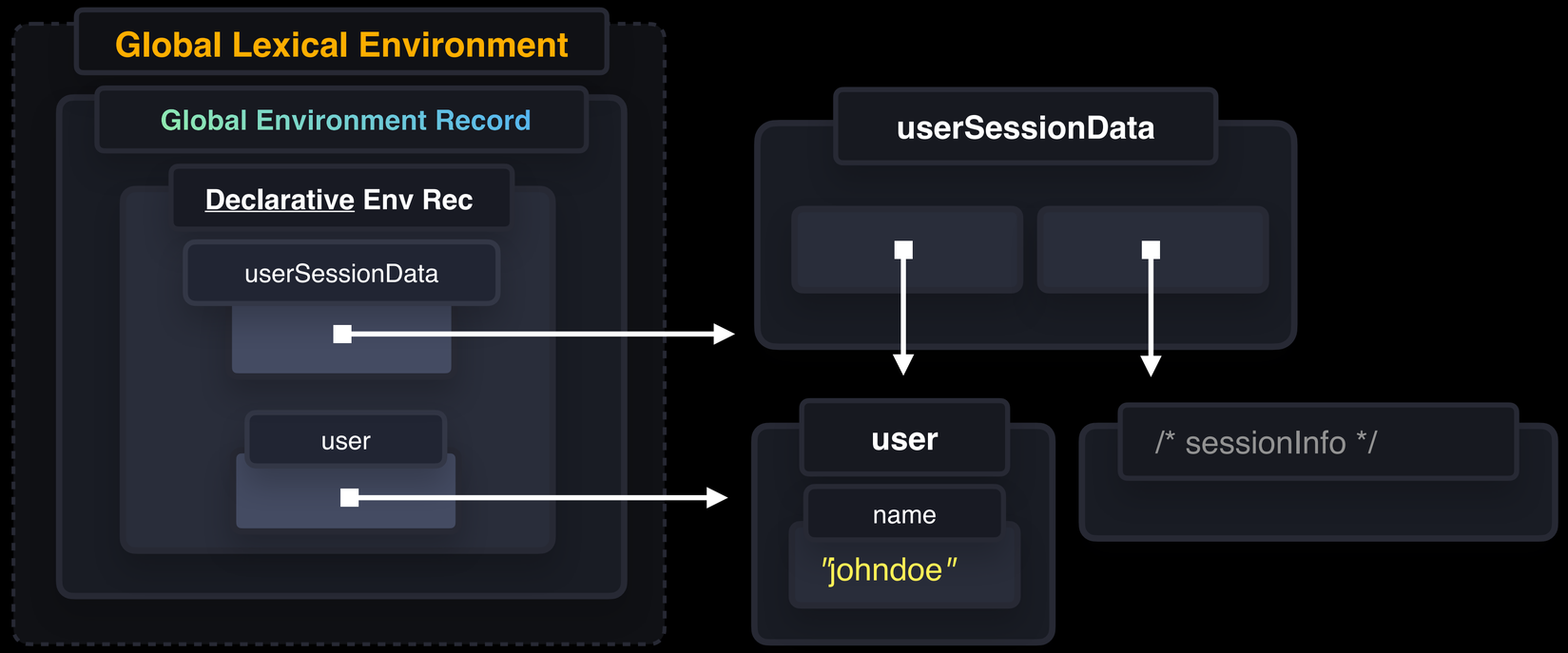
B Keys are weakly referenced, values are strongly referenced

C It is enumerable, allowing iteration over its elements

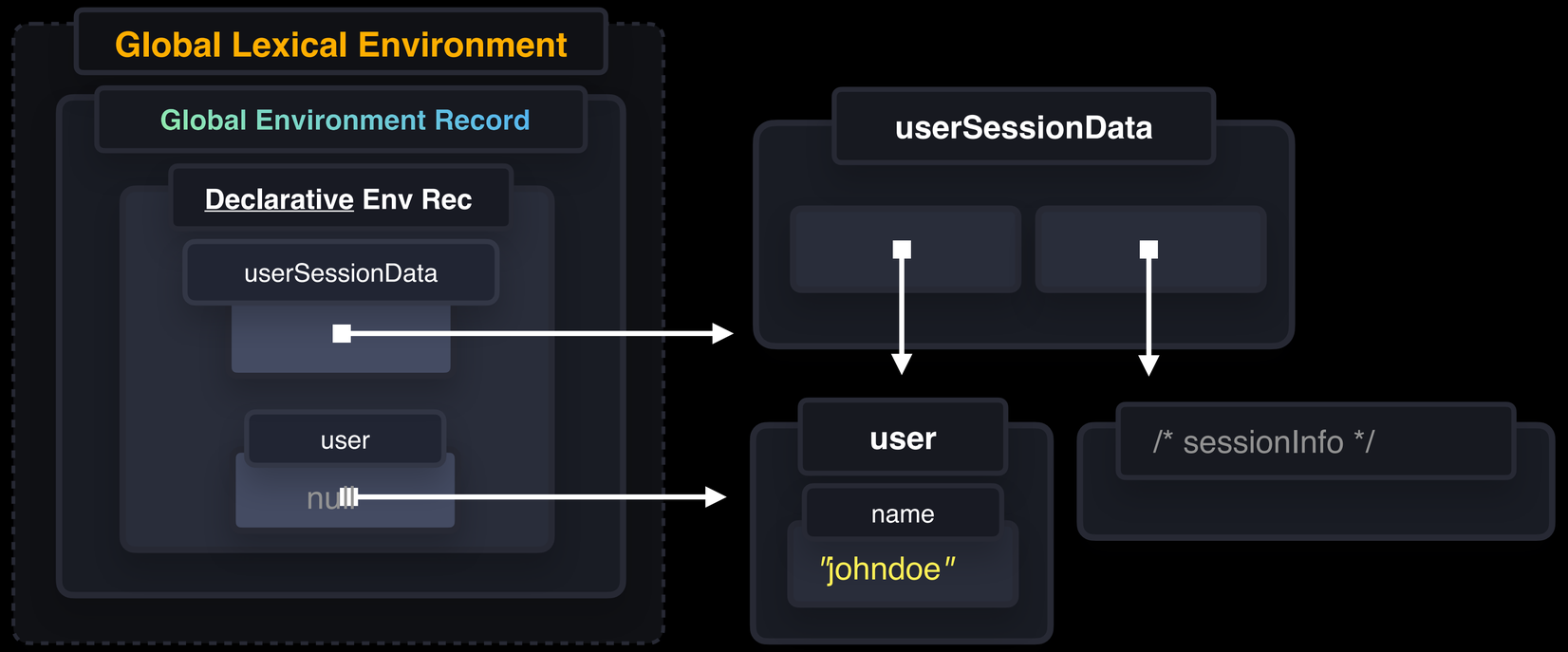
D It can have primitive data types as keys

E We can use the keys() method on a WeakMap, but not values()

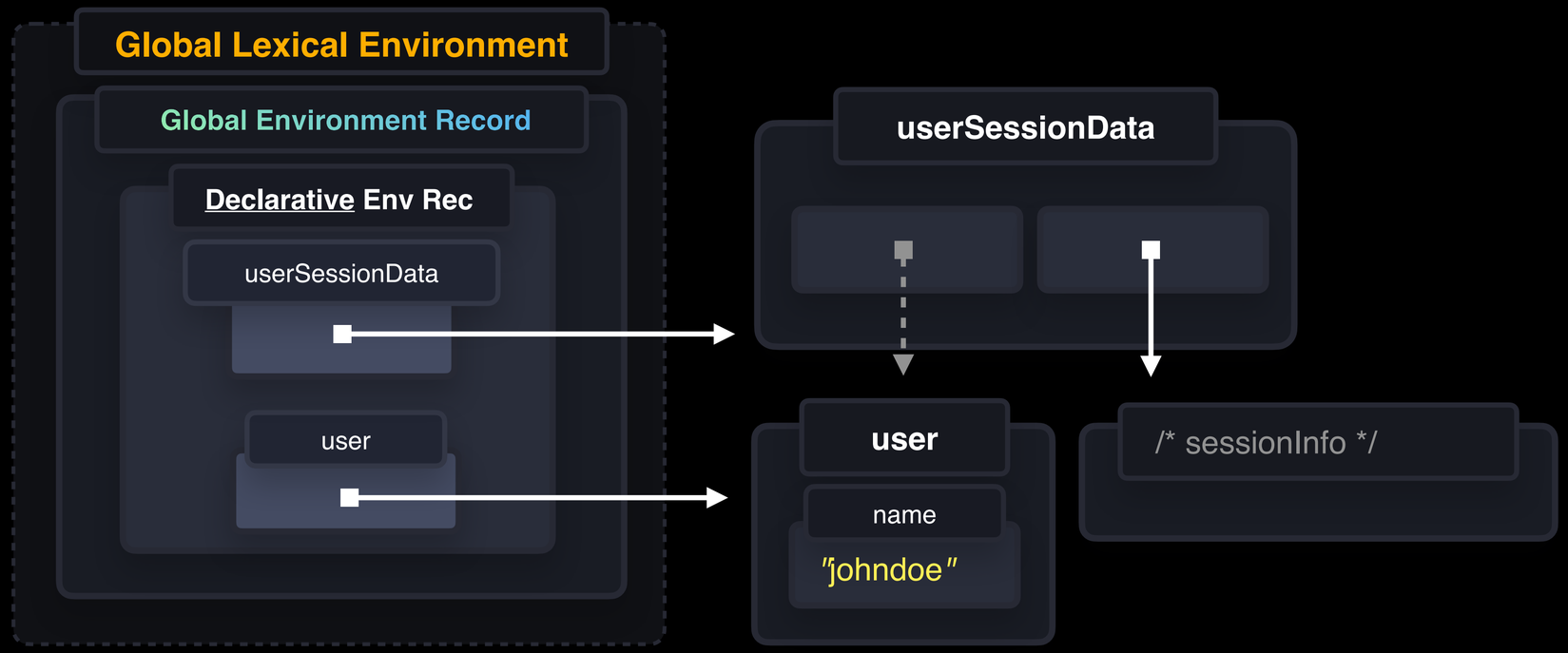
```
1  const user = { name: "johndoe" }
2  const userSessionData = new Map();
3
4  userSessionData.set(user, { /* sessionInfo */ });
```



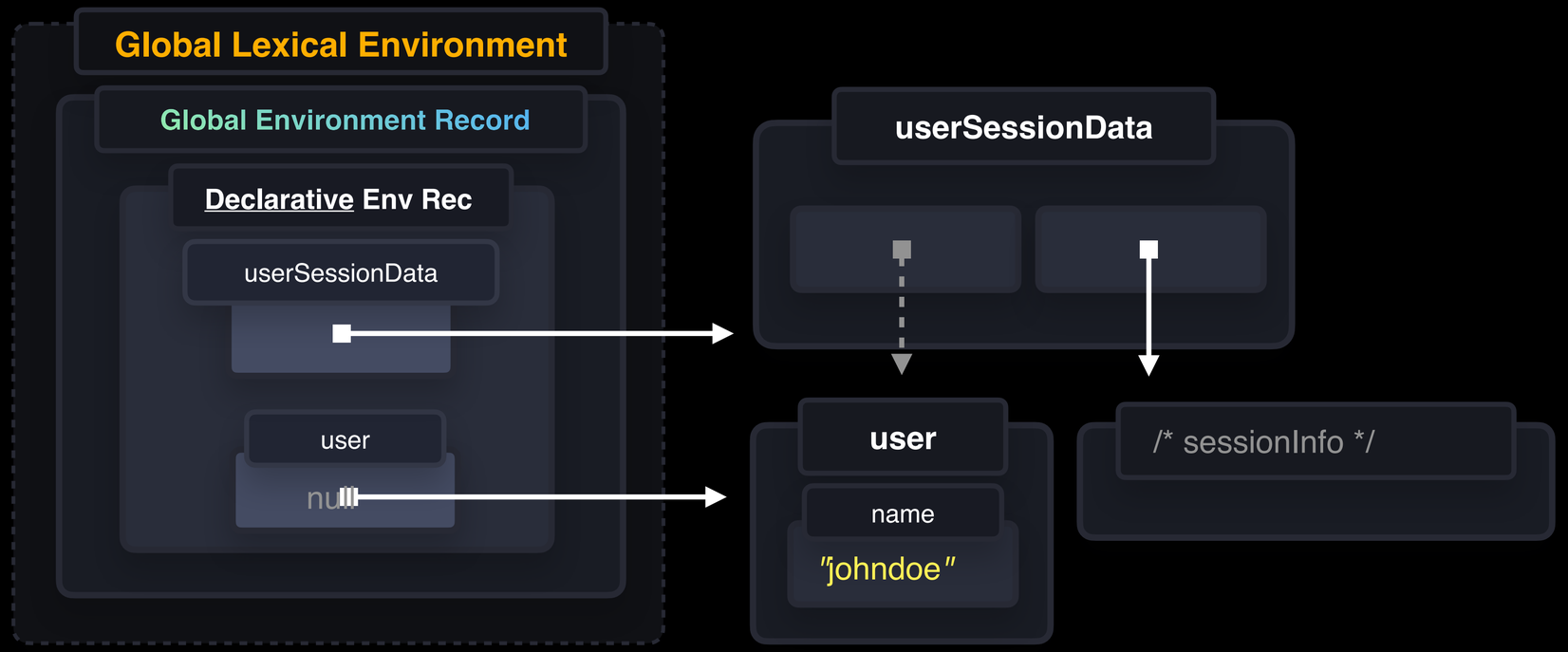
```
1  const user = { name: "johndoe" }
2  const userSessionData = new Map();
3
4  userSessionData.set(user,  { /* sessionInfo */ });
5  user = null
```



```
1  const user = { name: "johndoe" }
2  const userSessionData = new WeakMap();
3
4  userSessionData.set(user, { /* sessionInfo */ });
```



```
1  const user = { name: "johndoe" }
2  const userSessionData = new WeakMap();
3
4  userSessionData.set(user,  { /* sessionInfo */ });
5  user = null
```



Question 26

When will inner function be eligible for garbage collection?

```
function outer() {
  return function inner() {
    console.log("Inner function")
  }
}

const outerFunction = outer()
```

A By invoking outerFunction

B By explicitly setting outerFunction to null

C It is automatically garbage collected right after outer is called

D It depends on the JavaScript engine's garbage collection strategy

Question 26

When will inner function be eligible for garbage collection?

```
1  function outer() {
2    return function inner() {
3      console.log("Inner function")
4    }
5  }
6
7  const outerFunction = outer()
```

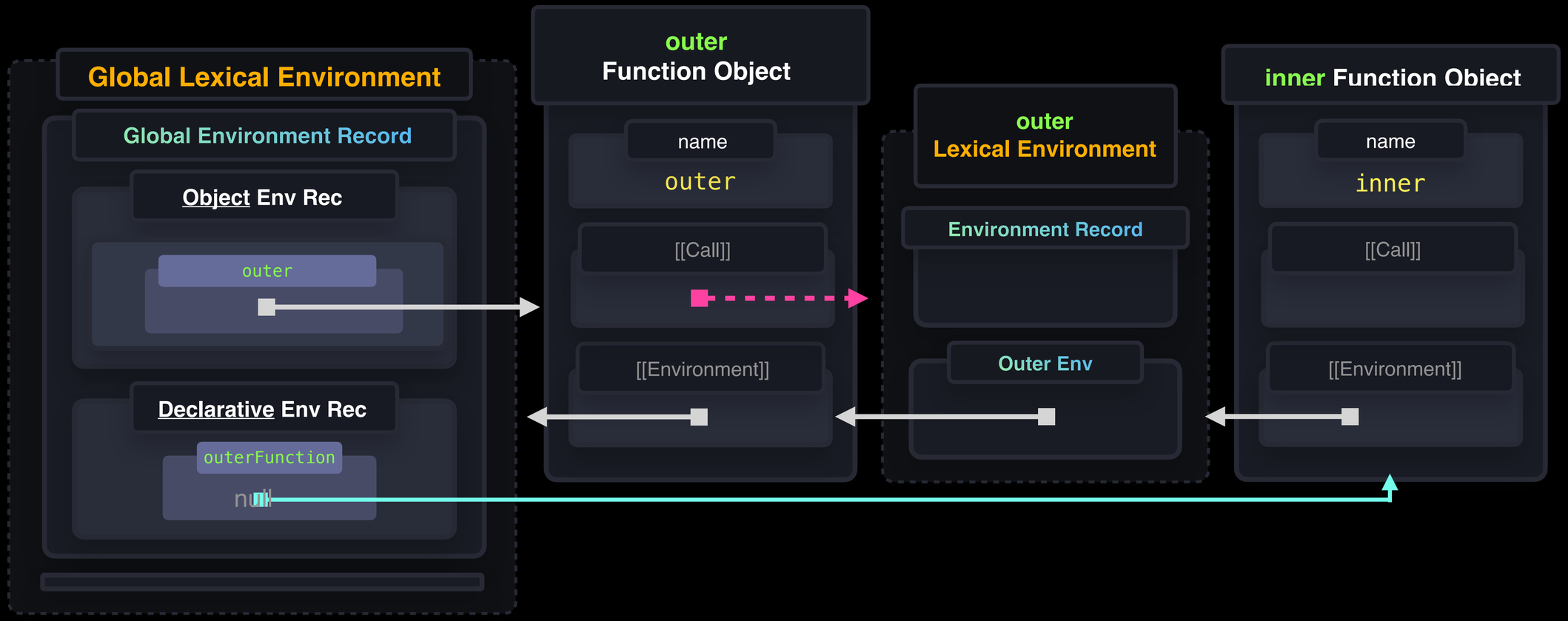
A By invoking outerFunction

B By explicitly setting outerFunction to null

C It is automatically garbage collected right after outer is called

D It depends on the JavaScript engine's garbage collection strategy

```
1  function outer() {
2    return function inner() {
3      console.log("Inner function")
4    }
5  }
6
7  const outerFunction = outer()
```



Question 27

Which statements are true regarding this code snippet?

```
1  let obj = { a: { b: 1 } };
2  let ref = obj.a;
3
4  obj = null;
```

A The object { b: 1 } will be garbage collected

B ref still references { b: 1 }, so it won't be garbage collected

C The entire obj object is retained in memory due to the reference in ref

D Setting obj to null frees all memory associated with it

Question 27

Which statements are true regarding this code snippet?

```
1  let obj = { a: { b: 1 } };
2  let ref = obj.a;
3
4  obj = null;
```

A The object { b: 1 } will be garbage collected

B ref still references { b: 1 }, so it won't be garbage collected

C The entire obj object is retained in memory due to the reference in ref

D Setting obj to null frees all memory associated with it

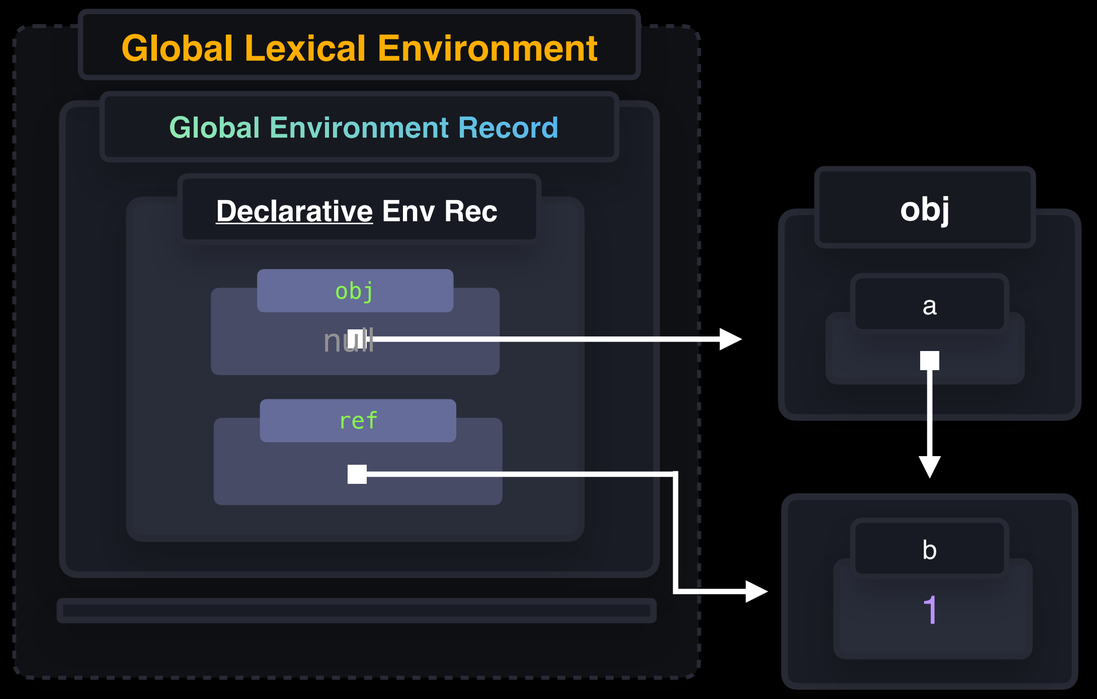
```
1  let obj = { a: { b: 1 } };
2  let ref = obj.a;
3
4  obj = null;
```

A The object { b: 1 } will be garbage collected

B ref still references { b: 1 }, so it won't be garbage collected

C The entire obj object is retained in memory due to the reference in ref

D Setting obj to null frees all memory associated with it



Question 28

You can get a list of all keys in a WeakMap using its keys method, but you can not get its values

A true

B false

Question 28

You can get a list of all keys in a WeakMap using its keys method, but you can not get its values

A true

B false

```
1   const user = { name: "johndoe" }
2   const userSessionData = new WeakMap();
3
4   userSessionData.set(
5     user,
6     { /* sessionInfo */ }
7   );
8
9   userSessionData.keys(); // TypeError
10  userSessionData.values(); // TypeError
11  [...userSessionData]; // TypeError
```

Question 29

When will each user be eligible for garbage collection?

```
1  function myFunc() {
2    for (let i = 0; i < 3; i++) {
3      const user = { name: "John Doe" };
4      return user;
5    }
6  }
```

A Immediately after each iteration.

B After the loop completes

C Only if explicitly set to **null** within the loop.

D When the function containing the loop finishes execution

Question 29

When will each user be eligible for garbage collection?

```
1  function myFunc() {
2    for (let i = 0; i < 3; i++) {
3      const user = { name: "John Doe" };
4      return user;
5    }
6  }
```

A Immediately after each iteration.

B After the loop completes

C Only if explicitly set to **null** within the loop.

D When the function containing the loop finishes execution

```
function myFunc() {
  for (let i = 0; i < 3; i++) {
    const user = { name: "John Doe" };
    return user;
  }
}
```

Question 30

What gets logged?

```
1  const obj = {
2    bar() {
3      console.log(this)
4    }
5  }
6
7  setTimeout(() => obj.bar(), 0);
8
9  queueMicrotask(() =>
10     delete obj.bar
11 });
```

A obj

B window

C setTimeout

D undefined

E TypeError

Question 30

What gets logged?

```
1  const obj = {
2    bar() {
3      console.log(this)
4    }
5  }
6
7  setTimeout(() => obj.bar(), 0);
8
9  queueMicrotask(() =>
10     delete obj.bar
11 });
```

A obj

B window

C setTimeout

D undefined

E TypeError

```
1   const obj = {
2     bar() {
3       console.log(this)
4     }
5   }
6
7   setTimeout(() => obj.bar(), 0);
8
9   queueMicrotask(() =>
10    delete obj.bar
11  });
```

Modules

Question 31

Match each module loading mechanism with its correct characteristic

1	CommonJS	A	Focuses on client-side asynchronous loading, allowing modules to be fetched and executed in a non-blocking manner
2	AMD	B	supports various environments, accommodating both server and client-side usage
3	UMD	C	used mainly in server-side contexts, this approach loads modules sequentially
4	ESM	D	uses static syntax for imports/exports

Question 31

Match each module loading mechanism with its correct characteristic

1	CommonJS	C	used mainly in server-side contexts, this approach loads modules sequentially
2	AMD	A	focuses on client-side asynchronous loading, allowing modules to be fetched and executed in a non-blocking manner
3	UMD	B	supports various environments, accommodating both server and client-side usage
4	ESM	D	uses static syntax for imports/exports

1 CommonJS

C used mainly in server-side contexts, this approach loads modules sequentially

```
const sum = require("./sum");

const value = sum(2, 3);

module.exports = { value };
```

- Modules are loaded synchronously, typically on the server-side.
- Uses require() for importing and module.exports for exporting.
- The primary module system for Node.js.

- Requires bundling and transpilation for use in browsers.
- Can cause performance issues when loading modules on the client-side.

2 AMD

A focuses on client-side asynchronous loading, allowing modules to be fetched and executed in a non-blocking manner

```
define(‘main', ['sum', function(sum) {
  const value = sum(2, 3);
  return { value };
});
```

- Modules and their dependencies are loaded asynchronously.
- Modules can be loaded at runtime as needed.
- Uses **define** for defining modules and **require** for loading modules.

- More complex syntax compared to CommonJS or ESM.
- Not commonly used in Node.js environments.

3 UMD

B supports various environments, accommodating both server and client-side usage

```
(function(root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define([], factory);
    } else if (typeof module === 'object' && module.exports) {
        // Node.js/ CommonJS
        module.exports = factory();
    } else {
        // Browser globals (root is typically `window`)
        root.sum = factory();
    }
}(typeof self !== 'undefined' ? self : this, function() {
    // Module definition and factory function
    return function sum(a, b) {
        return a + b;
    };
}));
```

- Works with both client-side and server-side JavaScript.
- Can switch between AMD and CommonJS environments, and also work in global scope.
- Ideal for libraries intended to be used in various environments.

- More complex than using either AMD or CommonJS alone.
- Can lead to larger file sizes due to handling multiple environments.

4 ESM

D supports various environments, accommodating both server and client-side usage

```
import sum from "./sum";

const value = sum(2, 3);

export default { value };
```

- Allows for static analysis, tree shaking, and more efficient bundling.
- Supports both synchronous and asynchronous loading.
- Directly supported by modern web browsers.
- Uses import and export keywords.

- Requires transpilation and bundling for legacy browser support.
- Requires changes to codebase when migrating from CommonJS.

Question 32

Put the logs in the correct order

```
import { createRequire } from "module";
const require =
    createRequire(import.meta.url);

require("./file1.js")
import "./file2.mjs"
import "./file3.mjs"

function getModule() {
    import("./file4.mjs")
    require("./file5.js")
}

getModule()
```

```
console.log("file1.js")
module.exports = {}
```

```
console.log("file2.mjs")
export default {}
```

```
await new Promise((res) =>
    setTimeout(() => res(), 0))

console.log("file3.mjs")
export default {}
```

```
console.log("file4.mjs")
export default {}
```

```
console.log("file5.js")
module.exports = {}
```

"file1.js"

"file2.js"

"file3.js"

"file4.js"

"file5.js"

Question 32

Put the logs in the correct order

```
import { createRequire } from "module";
const require =
    createRequire(import.meta.url);

require("./file1.js")
import "./file2.mjs"
import "./file3.mjs"

function getModule() {
    import("./file4.mjs")
    require("./file5.js")
}

getModule()
```

```
console.log("file1.js")
module.exports = {}
```

```
console.log("file2.mjs")
export default {}
```

```
await new Promise((res) =>
    setTimeout(() => res(), 0))

console.log("file3.mjs")
export default {}
```

```
console.log("file4.mjs")
export default {}
```

```
console.log("file5.js")
module.exports = {}
```

"file2.js"

"file3.js"

"file1.js"

“file5.js"

“file4.js"

Question 33

Which of the following statements correctly describe the differences between `require` and `import`

A	`require` can be called conditionally, while the `import` statement cannot
B	`import` statements are hoisted, but `require` calls are not
C	`require` synchronously loads modules, while `import` can load modules asynchronously
D	`import` allows for static analysis and tree shaking, but `require` does not

Question 33

Which of the following statements correctly describe the differences between `require` and `import`

A `require` can be called conditionally, while the `import` statement cannot

B `import` statements are hoisted, but `require` calls are not

C `require` synchronously loads modules, while `import` can load modules asynchronously

D `import` allows for static analysis and tree shaking, but `require` does not

Miscellaneous

Question 34

What gets logged?

```
let number = 0;
console.log(number++);
console.log(++number);
console.log(number);
```

A 1 1 2

B 1 2 2

C 0 2 2

D 0 1 2

E 1 1 0

Question 34

What gets logged?

```
let number = 0;
console.log(number++);
console.log(++number);
console.log(number);
```

A 1 1 2

B 1 2 2

C 0 2 2

D 0 1 2

E 1 1 0

```
1  let number = 0;
2  console.log(number++);
3  console.log(++number);
4  console.log(number);
```

Question 35

What gets logged?

```
1  const a = 3;
2  const b = new Number(3);
3  const c = 3;
4
5  console.log(a == b);
6  console.log(a === b);
7  console.log(b === c);
8  console.log(a.toString() === b.toString());
```

A	true	false	true	false
B	false	false	true	false
C	true	false	false	true
D	false	true	true	true
E	false	true	true	false

Question 35

What gets logged?

```
1  const a = 3;
2  const b = new Number(3);
3  const c = 3;
4
5  console.log(a == b);
6  console.log(a === b);
7  console.log(b === c);
8  console.log(a.toString === b.toString);
```

A	true	false	true	false
B	false	false	true	false
C	true	false	false	true
D	false	true	true	true
E	false	true	true	false

```
1  const a = 3;
2  const b = new Number(3);
3  const c = 3;
4
5  console.log(a == b);
6  console.log(a === b);
7  console.log(b === c);
8  console.log(a.toString === b.toString);
```

c	true	false	true	true

Question 36

What gets logged?

```
1  const a = isNaN("5.2" + 2);
2  const b = isNaN(parseInt(a));
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

A	true	true	true	true
B	false	false	false	false
C	true	false	false	true
D	true	false	false	false
E	false	true	true	true

Question 36

What gets logged?

```
1  const a = isNaN("5.2" + 2);
2  const b = isNaN(parseInt(a));
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

A	true	true	true	true
B	false	false	false	false
C	true	false	false	true
D	true	false	false	false
E	false	true	true	true

```
1  const a = isNaN(        "5.2" + 2       );
2  const b = isNaN(parseInt(a));
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = isNaN(        "5.22"    );
2  const b = isNaN(parseInt(a));
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = false;
2  const b = isNaN(parseInt(a));
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = false;
2  const b = isNaN(parseInt(false));
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = false;
2  const b = isNaN(NaN);
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = false;
2  const b = true;
3  const c = isNaN(parseFloat(a));
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = false;
2  const b = true;
3  const c = true;
4  const d = isNaN("1 * 2" * 2);
5
6  console.log(a, b, c, d);
```

```
1  const a = false;
2  const b = true;
3  const c = true;
4  const d = true;
5
6  console.log(a, b, c, d);
```

Question 37

Match the equivalent values

1	"ff"	A	255..toString(16)
2	"11111111"	B	255["toString"](2)
3	SyntaxError	C	255.toString(10)
4	"255"	D	255["toString"](10)

Question 37

Match the equivalent values

1	"ff"	A	255..toString(16)
2	"11111111"	B	255["toString"](2)
3	SyntaxError	C	255.toString(10)
4	"255"	D	255["toString"](10)

1	"ff"	A	255..toString(16)
2	"11111111"	B	255["toString"](2)
3	SyntaxError	C	255.toString(10)
4	"255"	D	255["toString"](10)

Question 38

Which method(s) will return the value "Hello World"?

```
1  const myMap = new Map();
2  const myFunc = () => "greeting";
3
4  myMap.set(myFunc, "Hello World”);
```

A myMap.get("greeting")

B myMap.get(myFunc)

C myMap.get(() => "greeting")

D myMap.get(myFunc())

E None of the above

Question 38

Which method(s) will return the value "Hello World"?

```
1  const myMap = new Map();
2  const myFunc = () => "greeting";
3
4  myMap.set(myFunc, "Hello World”);
```

A myMap.get("greeting")

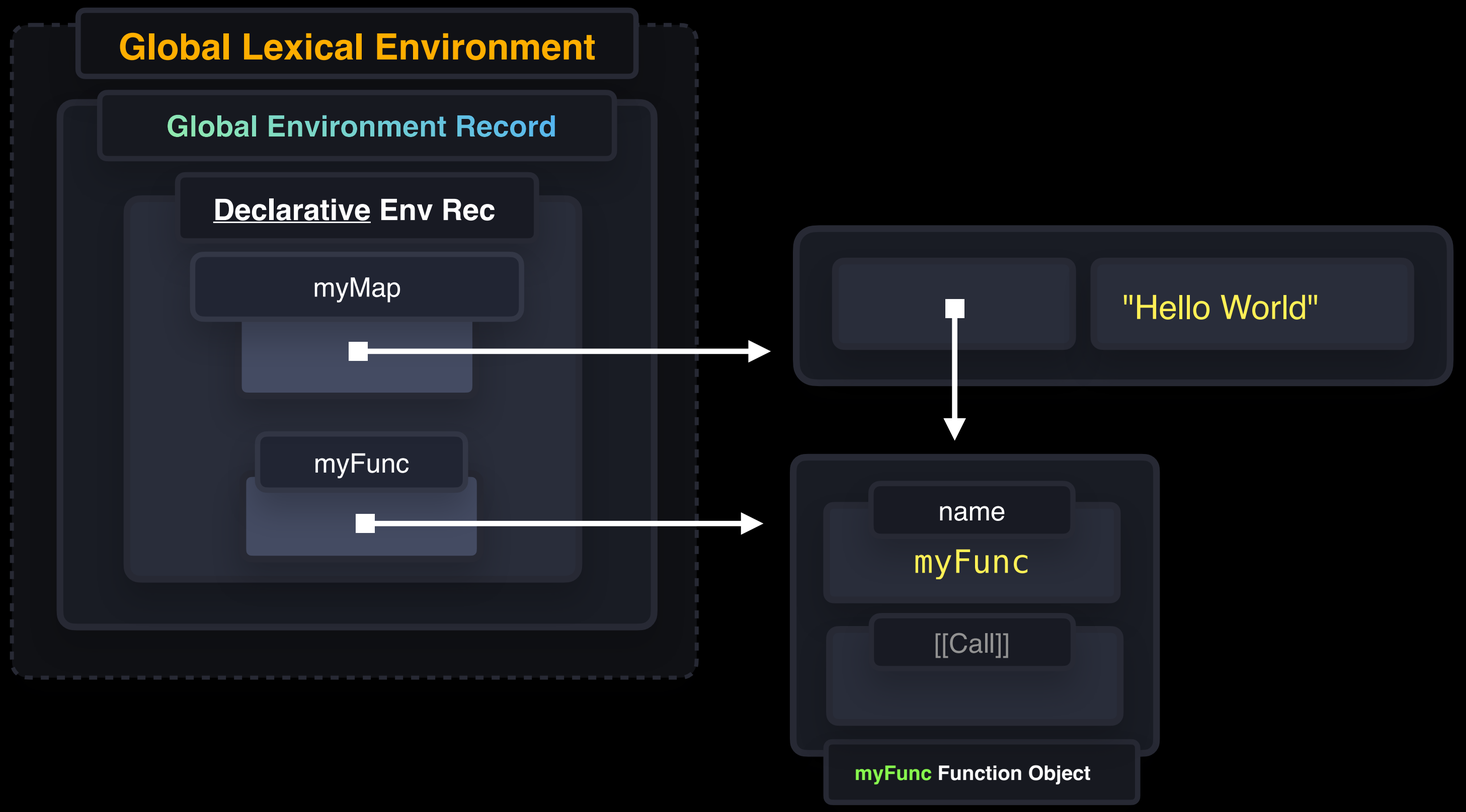
B myMap.get(myFunc)

C myMap.get(() => "greeting")

D myMap.get(myFunc())

E None of the above

```
1   const myMap = new Map();
2   const myFunc = () => "greeting";
3
4   myMap.set(myFunc, "Hello World");
```



Question 39

What does the person object look like after executing this script?

```
const person = {
  name: "Jane",
  address: { city: "Amsterdam" }
};

Object.freeze(person);

const personProxy = new Proxy(person, {
   set(...args) {
     Reflect.set(…args)
   }
})

personProxy.name = "Sarah";
personProxy.address.city = "Paris"

console.log(person)
```

A

```
{
  name: "Jane",
  address: { city: "Amsterdam" }
}
```

B

```
{
  name: “Sarah",
  address: { city: "Amsterdam" }
}
```

C

```
{
  name: "Jane",
  address: { city: "Paris" }
}
```

D

```
{
  name: “Sarah",
  address: { city: "Paris" }
}
```

E

TypeError

Question 39

What does the person object look like after executing this script?

```
1  const person = {
2    name: "Jane",
3    address: { city: "Amsterdam" }
4  };
5
6  Object.freeze(person);
7
8  const personProxy = new Proxy(person, {
9     set(...args) {
10      Reflect.set(…args)
11    }
12 })
13
14 personProxy.name = "Sarah";
15 personProxy.address.city = "Paris"
16
17 console.log(person)
```

A

```
{
  name: "Jane",
  address: { city: "Amsterdam" }
}
```

B

```
{
  name: “Sarah",
  address: { city: "Amsterdam" }
}
```

C

```
{
  name: "Jane",
  address: { city: “Paris" }
}
```

D

```
{
  name: “Sarah",
  address: { city: "Paris" }
}
```

E

TypeError

```javascript
const person = {
  name: "Jane",
  address: { city: "Amsterdam" }
};

Object.freeze(person);

const personProxy = new Proxy(person, {
   set(obj, prop, value) {
     Reflect.set(...arguments)
   }
})

personProxy.name = "Sarah";
personProxy.address.city = "Paris"

console.log(person)
```

Question 40

What will be the output of the following code if executed in a module?

```
const obj = Object.freeze({
  name: "John"
})

obj.name = "Jane"

console.log(obj.name);
```

A Jane

B John

C TypeError

D undefined

E ReferenceError

Question 40

What will be the output of the following code if executed in a module?

```
1  const obj = Object.freeze({
2      name: "John"
3  })
4
5  obj.name = "Jane"
6
7  console.log(obj.name);
```

A Jane

B John

C TypeError

D undefined

E ReferenceError

```
1  const obj = Object.freeze({
2      name: "John"
3  })
4
5  obj.name = "Jane"
6
7  console.log(obj.name);
```

```
1  "use strict"
2
3  const obj = Object.freeze({
4      name: "John"
5  })
6
7  obj.name = "Jane"
8
9  console.log(obj.name);
```

Question 41

Which method should we choose so it logs `{ name: "Lydia", age: 25 }`

```
const keys = ["name", "age"];
const values = ["Lydia", 25];

Object[???](keys.map((_, i) => {
  return [keys[i], values[i]]
}))
```

A entries

B values

C fromEntries

D forEach

E keys

Question 41

Which method should we choose so it logs `{ name: "Lydia", age: 25 }`

```
1  const keys = ["name", "age"];
2  const values = ["Lydia", 25];
3
4  Object[???](keys.map((_, i) => {
5    return [keys[i], values[i]]
6  }))
```

A entries

B values

C fromEntries

D forEach

E keys

```
1  const keys = ["name", "age"];
2  const values = ["Lydia", 25];
3
4  Object[???](keys.map((_, i) => {
5    return [keys[i], values[i]]
6  }))
```

A entries

B values

C fromEntries

D forEach

E keys

Question 42

What gets logged?

```
1  const array = [1, 2, 3, 4, 5];
2  array.splice(2);
3  array.concat(6);
4  array.slice(0, 1);
5  delete array[0]
6
7  console.log(array.length);
```

A 1

B 2

C 4

D 5

E 6

Question 42

What gets logged?

```
1  const array = [1, 2, 3, 4, 5];
2  array.splice(2);
3  array.concat(6);
4  array.slice(0, 1);
5  delete array[0]
6
7  console.log(array.length);
```

A 1

B 2

C 4

D 5

E 6

```
1  const array = [1, 2, 3, 4, 5];
2  array.splice(2); // array: [1, 2]
3  array.concat(6); // array: [1, 2]
4  array.slice(0, 1); // array: [1, 2]
5  delete array[0]; // array: [empty, 2]
6
7  console.log(array.length); // 2
```

Question 43

What gets logged?

```
1  const config ={
2      languages: [],
3      set language(lang) {
4          return this.languages.push(lang)
5      }
6  }
7
8  config.language = "Dutch";
9  console.log(config.language);
```

A ["Dutch"]

B []

C 0

D undefined

E ReferenceError

Question 43

What gets logged?

```
1  const config ={
2      languages: [],
3      set language(lang) {
4          return this.languages.push(lang)
5      }
6  }
7
8  config.language = "Dutch";
9  console.log(config.language);
```

A ["Dutch"]

B []

C 0

D undefined

E ReferenceError

```
1  const config = {
2      languages: [],
3      set language(lang) {
4          return this.languages.push(lang)
5      }
6  }
7
8  config.language = "Dutch";
9  console.log(config.language);
```

Question 44

What gets logged?

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};

const result = team.members.map(getInfo);
```

A ['Unknown','Unknown', 'Unknown', 'Unknown']

B [undefined,'', null, 'Unknown']

C ['Unknown','Boston', 'Unknown', 'Unknown']

D ['Boston','Unknown', 'Boston', 'Unknown']

E TypeError

Question 44

What gets logged?

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};

const result = team.members.map(getInfo);
```

A ['Unknown','Unknown', 'Unknown', 'Unknown']

B [undefined,'', null, 'Unknown']

C ['Unknown','Boston', 'Unknown', 'Unknown']

D ['Boston','Unknown', 'Boston', 'Unknown']

E TypeError

Nullish Coalescing Operator ??

Returns its right-hand operand when its left-hand operand is `null` or `undefined`

Logical AND &&

Returns the value of the first operand if it is falsy

Logical OR ||

Returns the value of the first operand if it is truthy

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

(undefined ?? "Boston") || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

“Boston" || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

“Boston"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

```
("" ?? "Boston") || "Unknown"
```

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

"" || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

"Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

(null ?? "Boston") || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

"Boston" || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

"Boston"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

(undefined ?? undefined) || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

undefined || "Unknown"

```
const team = {
  members: [
    { info: { street: undefined, city: "Boston" } },
    { info: { street: "", city: "Boston" } },
    { info: { street: null, city: "Boston" } },
    null
  ]
};

const getInfo = (mem) => {
  const info = mem?.info;
  return (info?.street ?? info?.city) || "Unknown";
};
```

"Unknown"

Question 45

Match the code to the correct error it would throw

1	`new Array(-1)`	A	ReferenceError
2	`25.toString()`	B	TypeError
3	`[].reduce((acc, cur) => acc + cur);`	C	SyntaxError
4	`const { a: b = c } = { a: undefined, c: 12 };`	D	RangeError
5	`const a = { b: 1 }; console.log((a ?? 1) ll b);`	E	No error gets thrown

Question 45

Connect the code to the correct error it would throw

1	`new Array(-1)`
2	`25.toString()`
3	`[].reduce((acc, cur) => acc + cur);`
4	`const { a: b = c } = { a: undefined, c: 12 };`
5	`const a = { b: 1 }; console.log((a ?? 1) ll b);`

D	RangeError
C	SyntaxError
B	TypeError
A	ReferenceError
E	No error gets thrown

Question 46

Match the date methods with the correct values when invoked on new Date()

1	toDateString()	A	Sun Nov 19 2023
2	toISOString()	B	2023-11-19T23:11:08.263Z
3	toLocaleDateString()	C	Sun Nov 19 2023 22:11:08 GMT+0100 (Central European Standard Time)
4	toUTCString()	D	Sun, 19 Nov 2023 21:11:08 GMT
5	toString()	E	11/19/2023

Question 46

Match the date methods with the correct values when invoked on new Date()

1	toDateString()
2	toISOString()
3	toLocaleDateString()
4	toUTCString()
5	toString()

A	Sun Nov 19 2023
B	2023-11-19T23:11:08.263Z
E	11/19/2023
D	Sun, 19 Nov 2023 21:11:08 GMT
C	Sun Nov 19 2023 22:11:08 GMT+0100 (Central European Standard Time)

1 toDateString()

Converts the date portion of a Date object into a readable string, ignoring the time part.

2 toISOString()

Converts a Date object to a string using the ISO 8601 format

3 toLocaleDateString()

Converts the date portion of a Date object into a string, using locale-specific formatting.

4 toUTCString()

Converts a Date object to a string, using the UTC time zone.

5 toString()

Converts a Date object to a string, typically in a default format specific to the environment's locale.

Question 47

What gets logged?

```
const {
  a = 'default',
  b = 'default',
  c = 'default',
  d = 'default',
} = {
  a: null,
  b: undefined,
  c: false,
  d: 0
};

console.log(a, b, c, d);
```

A null undefined false 0

B null "default" false 0

C "default" "default" "default" "default"

D "default" "default" false 0

Question 47

What gets logged?

```
1   const {
2     a = 'default',
3     b = 'default',
4     c = 'default',
5     d = 'default',
6   } = {
7     a: null,
8     b: undefined,
9     c: false,
10    d: 0
11  };
12
13  console.log(a, b, c, d);
```

A null undefined false 0

B null "default" false 0

C "default" "default" "default" "default"

D "default" "default" false 0

Default parameters used if no value or undefined is passed.

```
const {
  a = 'default',
  b = 'default',
  c = 'default',
  d = 'default',
} = {
  a: null,
  b: undefined,
  c: false,
  d: 0
};

console.log(a, b, c, d);
```

Question 48

What gets logged?

```
1  const symbolOne = Symbol.for("key");
2  const symbolTwo = Symbol(“key");
3  const symbolThree = Symbol.for("key");
4
5  console.log(symbolOne === symbolTwo);
6  console.log(symbolTwo === symbolThree);
7  console.log(symbolOne === symbolThree);
8  console.log(symbolThree === Symbol("key"));
```

A	true	true	true	true
B	false	false	false	false
C	true	true	false	true
D	false	false	true	false
E	false	true	false	true

Question 48

What gets logged?

```
1  const symbolOne = Symbol.for("key");
2  const symbolTwo = Symbol("key");
3  const symbolThree = Symbol.for("key");
4
5  console.log(symbolOne === symbolTwo);
6  console.log(symbolTwo === symbolThree);
7  console.log(symbolOne === symbolThree);
8  console.log(symbolThree === Symbol("key"));
```

A	true	true	true	true
B	false	false	false	false
C	true	true	false	true
D	false	false	true	false
E	false	true	false	true

```
Symbol.for("key");
```

Checks if a symbol with the key "key" already exists in the global symbol registry.
If yes, it returns the existing symbol, otherwise it creates a new symbol and adds it to the global registry

```
Symbol("key");
```

Always creates a new, unique symbol that is different from any other symbol

```
1  const symbolOne = Symbol.for("key");
2  const symbolTwo = Symbol("key");
3  const symbolThree = Symbol.for("key");
4
5  console.log(symbolOne === symbolTwo);
6  console.log(symbolTwo === symbolThree);
7  console.log(symbolOne === symbolThree);
8  console.log(symbolThree === Symbol("key"));
```

D	false	false	true	false

Question 49

What gets logged?

```
function compareUsers(user1, user2 = user) {
    console.log(user1 === user2);
}

const user = { name: "Lydia" };
compareUsers(user, { ...user });
compareUsers(user);
compareUsers({ name: "Lydia" }, { name: "Lydia" })
```

A	true	true	false
B	false	true	false
C	false	false	false
D	true	false	false
E	ReferenceError		

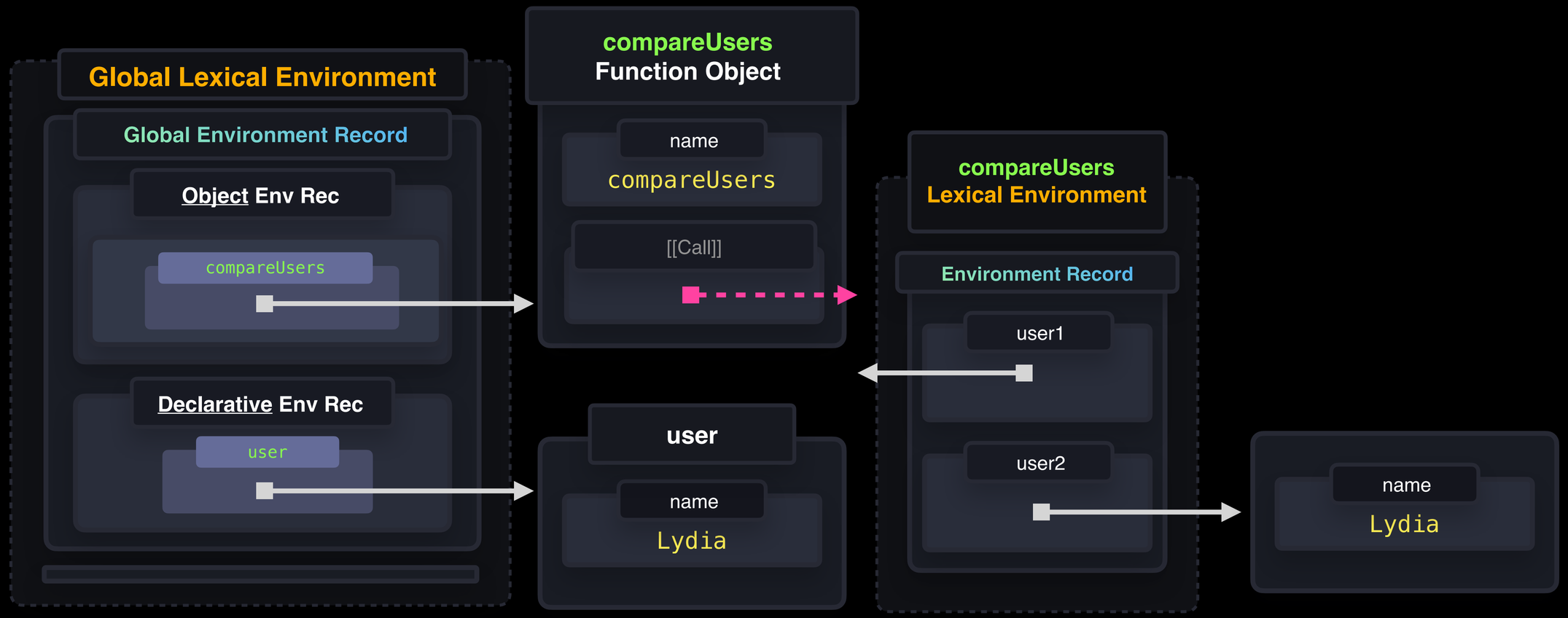
Question 49

What gets logged?

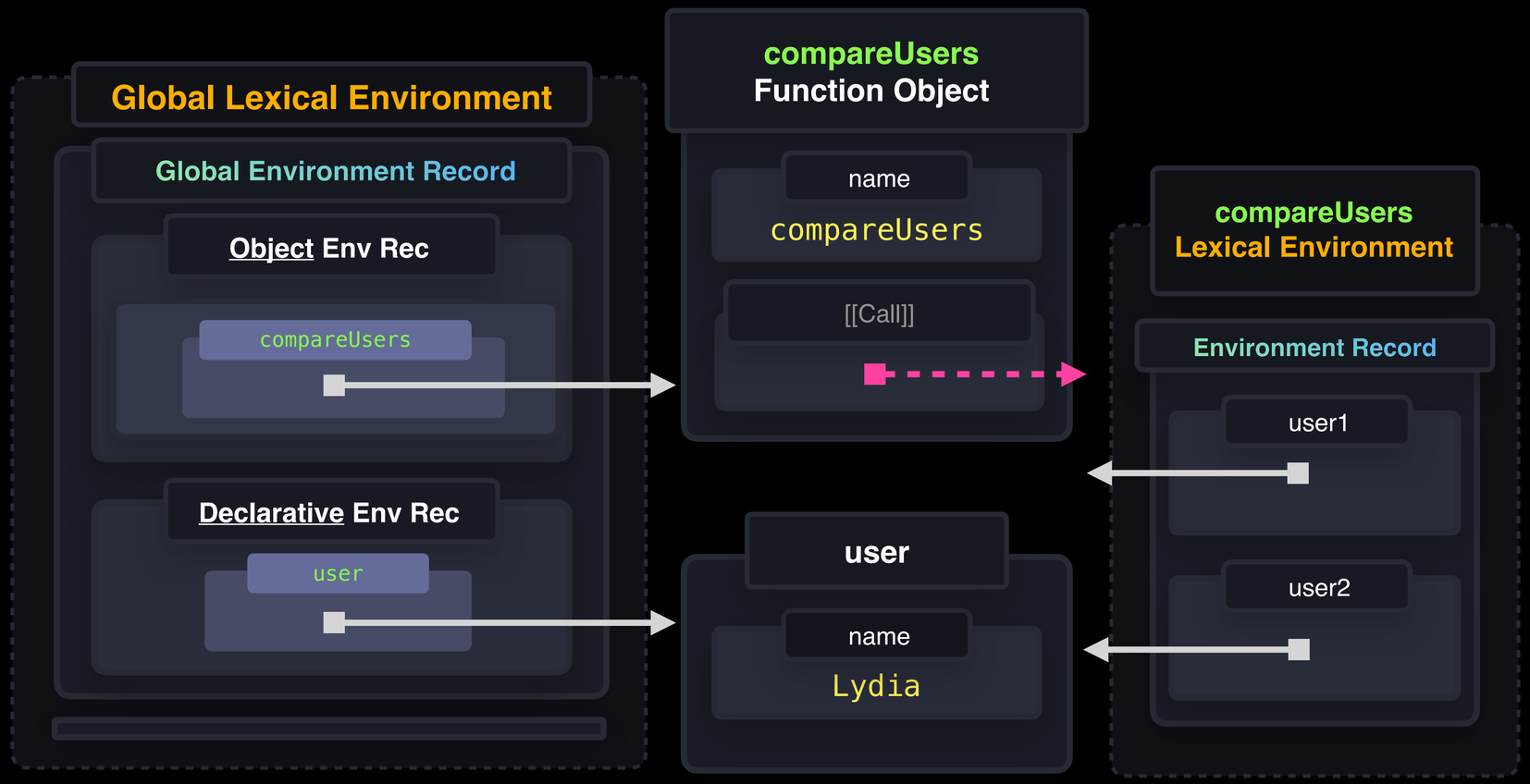
```
function compareUsers(user1, user2 = user) {
    console.log(user1 === user2);
}

const user = { name: "Lydia" };
compareUsers(user, { ...user });
compareUsers(user);
compareUsers({ name: "Lydia" }, { name: "Lydia" })
```

A	true	true	false
B	false	true	false
C	false	false	false
D	true	false	false
E	ReferenceError		

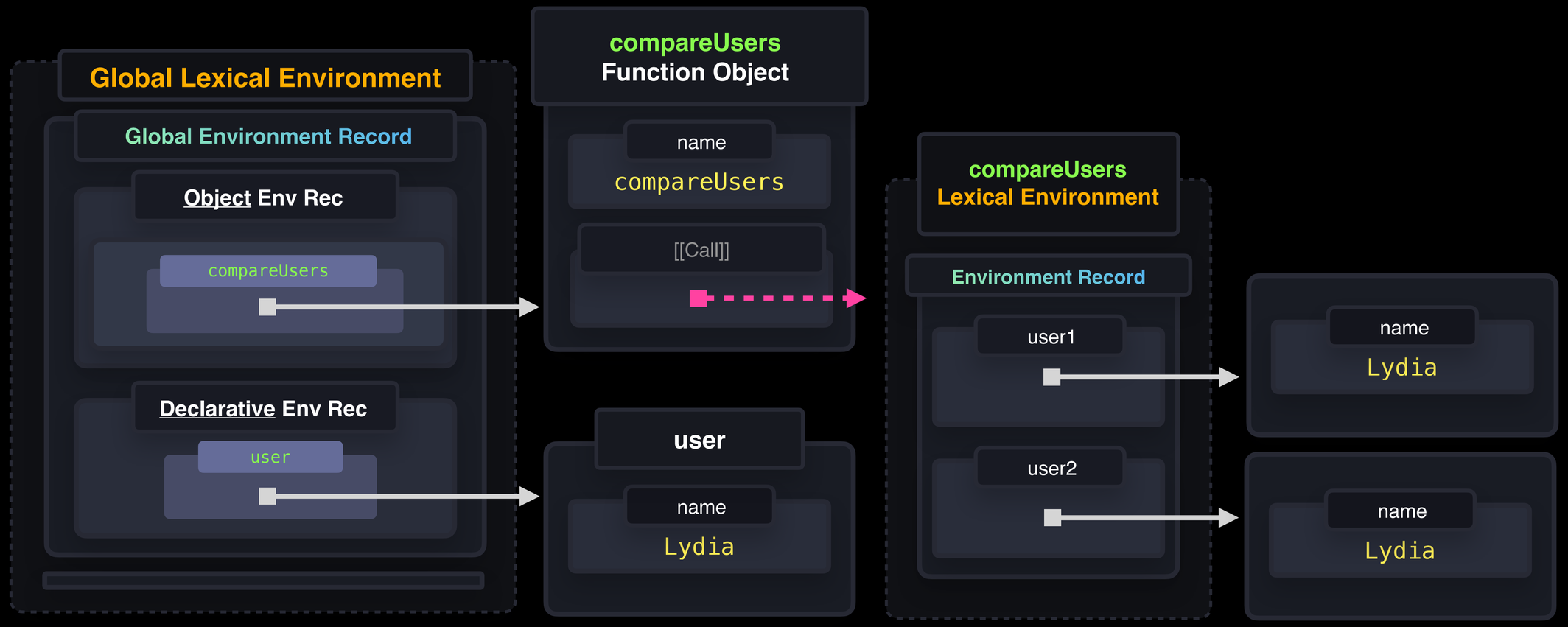
```
compareUsers(user, { ...user });
```



```
compareUsers(user);
```



```
compareUsers({ name: "Lydia" }, { name: "Lydia" });
```



Question 50

What gets logged?

```
function thankYouTag(strings, greeting) {
    console.log(strings, greeting)
}

const greeting = 'Thank you for coming to my workshop!';
thankYouTag`This is the last question! ${greeting}`
```

A
['This is the last question! ', ''],
'Thank you for coming to my workshop!'

B
'This is the last question! Thank you for coming to my workshop!'
'Thank you for coming to my workshop!'

C
'This is the last question! ',
'Thank you for coming to my workshop!'

D
['This is the last question! '],
['Thank you for coming to my workshop!']

E
['This is the last question! ', ''],
['Thank you for coming to my workshop!', ‘’]

Question 50

What gets logged?

```
function thankYouTag(strings, greeting) {
    console.log(strings, greeting)
}

const greeting = 'Thank you for coming to my workshop!';
thankYouTag`This is the last question! ${greeting}`
```

A ['This is the last question! ', ''],
'Thank you for coming to my workshop!'

B 'This is the last question! Thank you for coming to my workshop!'
'Thank you for coming to my workshop!'

C 'This is the last question! ',
'Thank you for coming to my workshop!'

D ['This is the last question! '],
['Thank you for coming to my workshop!']

E ['This is the last question! ', ''],
['Thank you for coming to my workshop!', ‘’]

```
function thankYouTag(arrayOfStrings, firstExpression, secondExpression, ...) {
  console.log(
    arrayOfStrings,          // ['This is the last question! ", " This is another part of the", '"']
    firstExpression,         // Thank you for coming!
    secondExpression         // string
  )
}
thankYouTag`This is the last question! ${greeting} This is another part of the ${"string"}`
```

```
function thankYouTag(strings, greeting) {
  console.log(
    strings,          // ['This is the last question! ', '']
    greeting,         // Thank you for coming to my workshop!
  )
}
const greeting = 'Thank you for coming to my workshop!';
thankYouTag`This is the last question! ${greeting}`
```

JavaScript Quiz