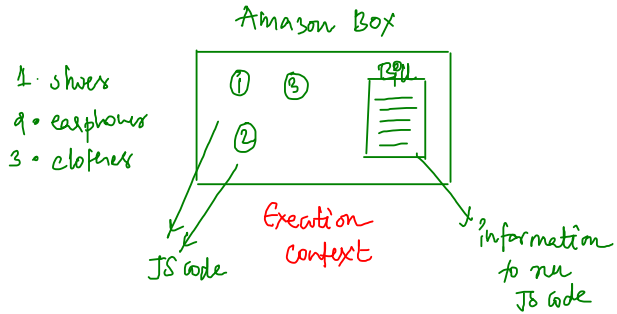
* Execution context :

→ It is like a black box, where a piece of JS code is executed, It contains all the information for the code to be executed.



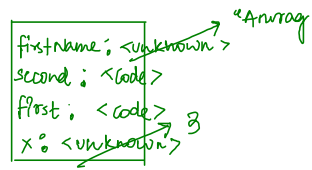
Information :

1. variables
   - → let, const, var
   - → function declarations
   - → parameters of function
2. Scope chain

2 phases → Creation phase
2 phases → execution phase

```
552  const firstName = "Anurag";
553
554  function second() {
555    let c = 2;
556    return c;
557  }
558
559  function first() {
560    let a = 1;
561    const b = second();
562    a = a + b;
563    return a;
564  }
565
566  const x = first();
567  console.log(x);
```

① Global Execution Context

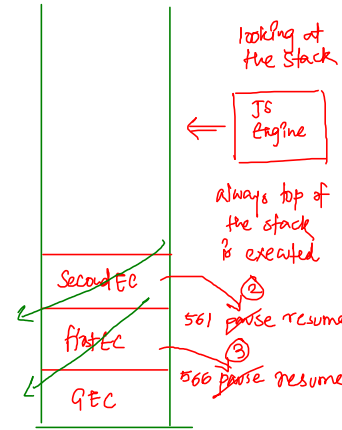


→ scan for variables and function declarations

(creation phase)

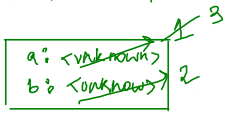
② we will run the GEC,

→ GEC is placed on top of stack.



* when first() is called

① EC is created



② place firstEC on stack

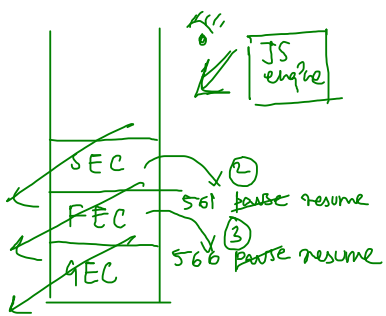
* when second is called,

① EC is created

② place secondEC on stack

return
⇒ remove EC from stack.

```
552  const firstName = "Anurag";
553
554  function second() {
555    let c = 2;
556    return c;
557  }
558
559  function first() {
560    let a = 1;
561    const b = second();
562    a = a + b;
563    return a;
564  }
565
566  const x = first();
567  console.log(x);
```



1. GEC created
2. GEC is paused at 566
3. first-EC is created and placed on stack
4. FEC is paused at 561
5. second EC is created & placed on stack
6. SEC is completed, removed from stack, returned (2)
7. Now as FEC is on top it resumes where it paused ⇒ 561 will recieve (2)
8. FEC is completed, removed from stack, returned (3)
9. Now as GEC is on top it resumes where it paused ⇒ 566 will recieve (3)
10. GEC is completed, removed from stack

★ Scope :

Global scope

```
const a = 5;
let firstName = "Anurag";
const year = 2023;

function test() {
  console.log("Inside test func", a, firstName, year);
}

test();
console.log("outside test func", a, firstName, year);
```

→ declared outside of all functions and if, else, for etc.. any block

→ declared in GEC

→ they are accessible every where in the code.

Function Scope

```
function second() {
  console.log(a);
  let c = 2;
  return c;
}

function first() {
  const a = 2;
  const b = second();
  console.log(c);
  a = a + b;
  return a;
}

const x = first();
console.log(x);
```

→ variables created in the function are accessible within that function only.

→ Reference Error: not defined

Block scope

```
const birthYear = 1988;
if (1981 <= birthYear && birthYear <= 1990) {
  const oldGen = true;
}
console.log(oldGen);
```

```
const birthYear = 1988;
let oldGen;
if (1981 <= birthYear && birthYear <= 1990) {
  oldGen = true;
}
console.log(oldGen);
```

(Correct it this way)

→ variables declared inside if/else/else if/for/while are accessible within those blocks only.

→ function / block scope are almost same.

```
const birthYear = 1988;
if (1981 <= birthYear && birthYear <= 1990) {
  var oldGen = true;
}
console.log(oldGen);

function test() {
  var firstName = "anurag";
}

test();
console.log(first);
```

★ block scope is applicable only for let, const but function scope applies for let, const, var. (this happens due to hoisting)

* Hoisting : (It is due to Execution context concept)

→ It makes variables & functions accessible in the code even before they are declared.

∀ Behind the scenes

Before execution, the creation phase is responsible for this Behavior.

	Are they hoisted ?	Initial value
Functions →	Yes	
var declarations →	Yes,	undefined
let, const declarations →	No,	< Temporal dead zone> <TDZ

Example :

```
619  test();
620  demo();
621  console.log(currYear);
622  console.log(example);
623
624  function test() {
625    console.log("Hi, How are you ?");
626  }
627
628  function demo() {
629    console.log("I am a demo function");
630  }
631
632  var currYear = 2024;
633  let example = "some random test";
```

① GEC Created

test : <code 624>
demo : <code 628>
currYear : undefined
example : <Tdz> → uninitialized

op: Hi, How are you ?
I am a demo function
undefined
(Error) → "Cannot access 'example' before initialization (Tdz)

② place GEC on stack

★ memory is allocated for var during creation phase only, but for and let it is allocated during execution.

★ test()

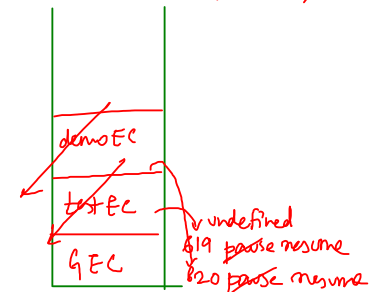
① testEC

② place on stack

★ demo()

① demoEC

② place on stack



Q1:

```
635  y = 3;
636  console.log(y);
637  var y = 100;
638  console.log(y);
```

→ let y = 100

① GEC Creation

② run the EC, place on stack

635. y = 3

→ Is y in the GEC? yes
so change value

636. 3 → read y from GEC

637. y = 100 → change y in GEC

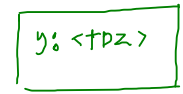
638. 100 → read y from GEC

* what if

637. let y = 100;

* var is hoisted, let/const are not hoisted.

① GEC Creation



② run the GEC

635. y = 3 (Error)

Is y in the GEC? yes

→ But you cannot change as it is in TDZ.

Q2:

```
640  function example() {
641    console.log(a);
642  }
643
644  console.log(a);
645  var a = 1;
646  example();
```

→ global scope

① GEC creation

example : <code 640>
a : undefined

1

② run,

644. C(a);

undefined

645. a = 1

646. example()

example EC

641. console.log(a) ⇒ 1

→ Is a present in my EC? No

→ Check a present in GEC?

yes

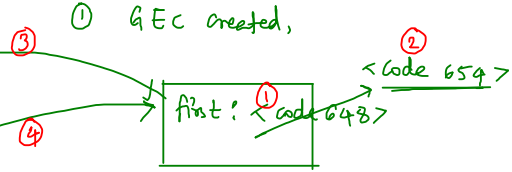
Q3 :

```
function first() {
  console.log(1);
}

first();

function first() {
  console.log(2);
}

first();
```



② run,

652 . 2

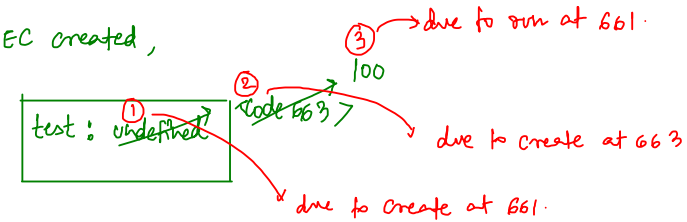
658 . 2

* If you have same function names, they will be replaced with the last / latest function (top → Bottom)

Q4 :

```
661  var test = 100;
662  console.log(test);
663  function test() {
664    console.log("Inside function test");
665  }
666  console.log(test);
```

① GEC created,



② run,

661. test = 100

662. 100

666. 100