①
```
if(true){
  var a=10;
  console.log(a);
}
```

We use blocks if we want to place multiple statements instead of a single statement.

②
```
{ var a=10;
  let b=20
  const c=30;
  console.log(a);
  console.log(b);
  console.log(c);
}
console.log(a);
console.log(b);
console.log(c);
```

» 10
20
30
10
▸ Uncaught ReferenceError: b is not defined.

We can access let and const variables only inside the block but var variable is in global scope, so it can be accessed anywhere.

③
```
var a=100;
{ var a=10;
  console.log(a); }
console.log(a);
```

SHADOWING

The value of a inside the block shadows the value of a outside the block.

Also a=10 modifies the value of a=100 because both points to the same memory location as a is in global scope.

③
```
const c=100;
{ var a=10;
  let b=20;
  const c=30;
  console.log(c);
}
console.log(c);
```

//

```
let c=100;
{ var a=10;
  let b=20
  let c=30;
  console.log(c);
}
console.log(c);
```

>> 30
100

c inside block is in block scope and c outside is in script scope.
So it prints different values seperately.

③ It behaves the same way inside a function.

```
Const c = 100;
function(){
  var a = 10;
    let  b = 20;
    const c = 30;
    console.log(c);
  }
console.log(c);
```

ILLEGAL SHADOWING

①
```
let a = 20;
{
 var a = 20;
 }
```
>> Uncaught Syntax Error: Identifier 'a' has already been declared
// We can't shadow let variable inside a block using var.

②
```
let a = 20;
{  let a = 100;
}
```
// This is valid.
We can shadow a let using a let.

③
```
var a = 20;
} let a = 20;
}
```
// This viceversa is perfectly fine

④ # If a variable is shadowing something, it shouldn't cross the boundary of its scope.

```
let a = 20;
{
  var a = 20;
 }
```
In a particular scope, let can't be redeclared. var shouldn't cross the boundary of its limit.

```
let a = 20;
function n(){
  var a = 20;
 }
```
This is perfectly fine. var is within its boundaries of function

```
const a = 20;
{
  const a = 20;
}
```

This is also fine. This is within its memory space.

⑤

```
const a = 20;
{
  const a = 100;
  {
    const a = 200;
    console.log(a);
  }
}
```

← will get access from the nearest a.

Block scope also follows lexical scope.

⑥ functions and arrow functions behaves the same in terms of scope.