var vs let

introduction to the `let` keyword for declaring block-scoped variables; and the dangers of scoping, such as the temporal dead zone
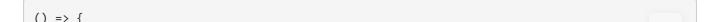
Variables declared with `var` have function scope. This means that they are accessible inside the function/block they are defined in. Take a look at the following code:

```
var guessMe = 2;
console.log("guessMe: "+guessMe);// A: guessMe is 2
( function() {
    console.log("guessMe: "+guessMe);// B: guessMe is undefined
    var guessMe = 5;
    console.log("guessMe: "+guessMe);// C: guessMe is 5
} )();
console.log("guessMe: "+guessMe);// D: guessMe is 2
```

Comment B may surprise you if you have not heard of hoisting. If a variable is declared using **var** inside a function, the Javascript engine treats them as if they are declared at the top of a functional scope. However, if that variable has been declared outside the function, it has a global scope regardless of where the actual declaration occurs. This is called **hoisting**.

```
() => {
    JAVASCRIPT_STATEMENTS;
    var guessMe = 5;
};
//accessing guessMe will give an error here
```

in the following form:

```
() => {
```

```
    var guessMe;
    JAVASCRIPT_STATEMENTS;

    guessMe = 5;
};
```

Variables declared with `var` are initialized to `undefined`. This is why the value of `guessMe` was `undefined` in comment `B`.

Variables declared with `let` have block scope. They are valid inside the block they are defined in.

```
// A: guessMe is undeclared
{
    // B: guessMe is uninitialized. Accessing guessMe throws an error
    //console.log(guessMe); <-This gives an error
    let guessMe = 5;
    console.log("guessMe: "+guessMe);// C: guessMe is 5
}
// D: guessMe is undeclared
```
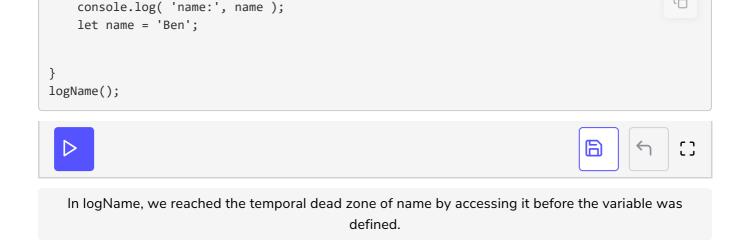
Comment `B` may surprise you again. Even though `let guessMe` is hoisted similarly to `var`, its value is not initialized to `undefined`. Retrieving uninitialized values throws a JavaScript error.

The area described by comment `B` is the *temporal dead zone* of variable `guessMe`.

```
function logAge() {
    console.log( 'age:', age );
    var age = 25;
}
logAge();
```

In `logAge`, we log `undefined`, as `age` is hoisted and initialized to `undefined`.

```
function logName() {
```

```
      console.log( 'name:', name );
      let name = 'Ben';


}
logName();
```

In logName, we reached the temporal dead zone of name by accessing it before the variable was defined.

You may find the temporal dead zone inconvenient at first sight. However, notice that the thrown error grasps your attention a lot better than a silent `undefined` value. Always be grateful for errors pointing out the obvious mistakes during development, as the same mistakes tend to be a lot more expensive once they are deployed to production.

The temporal dead zone exists even if a variable with the same name exists outside the scope of the dead zone.

```
let guessMe = 1;
console.log( 'guessMe: ', guessMe );// A: guessMe is 1
{
    // Temporal Dead Zone of guessMe
    //console.log( 'guessMe: ', guessMe ); <- This would give an error
    let guessMe = 2;
    console.log( 'guessMe: ', guessMe );// C: guessMe is 2
}
console.log( 'guessMe: ', guessMe );// D: guessMe is 1
```

For a complete reference, the temporal dead zone exists for `let`, `const`, and `class` declarations. It does not exist for `var`, `function`, and `function*` declarations.

Now, let's talk about the `const` keyword.