Looping code

- Why are loops useful?
  - Looping code example
  - With and without a loop
- Looping through a collection
  - The for...of loop
  - map() and filter()
- The standard for loop
  - Calculating squares
  - Looping through collections with a for loop
- Exiting loops with break
- Skipping iterations with continue
- while and do...while
- Implementing a launch countdown
- Filling in a guest list
- Which loop type should you use?
- Summary
- See also

Programming languages are very useful for rapidly completing repetitive tasks, from multiple basic calculations to just about any other situation where you've got a lot of similar items of work to complete. Here we'll look at the loop structures available in JavaScript that handle such needs.

Prerequisites:	An understanding of HTML and the fundamentals of CSS, familiarity with JavaScript basics as covered in previous lessons.

Learning outcomes:	• Understand the purpose of loops — a code structure that allows you to do something very similar many times without repeating the same code for each iteration. • General loop types such as `for` and `while`. • Looping through collections with constructs like `for...of` and `map()`. • Breaking out of loops and continuing.

Why are loops useful?

Loops are all about doing the same thing over and over again. Often, the code will be slightly different each time round the loop, or the same code will run but with different variables.

Looping code example

Suppose we wanted to draw 100 random circles on a canvas element (press the *Update* button to run the example again and again to see different random sets):

```
<button>Update</button> <canvas></canvas>
```

```
html {
  width: 100%;
  height: inherit;
  background: #dddddd;
}

canvas {
  display: block;
}

body {
  margin: 0;
}

button {
  position: absolute;
  top: 5px;
  left: 5px;
```

```
}
```

Here's the JavaScript code that implements this example:

```
const btn = document.querySelector("button");
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");

canvas.width = document.documentElement.clientWidth;
canvas.height = document.documentElement.clientHeight;

function random(number) {
  return Math.floor(Math.random() * number);
}

function draw() {
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  for (let i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = "rgb(255 0 0 / 50%)";
    ctx.arc(
      random(canvas.width),
      random(canvas.height),
      random(50),
      0,
      2 * Math.PI,
    );
    ctx.fill();
  }
}

btn.addEventListener("click", draw);
```

With and without a loop

You don't have to understand all the code for now, but let's look at the part of the code that actually draws the 100 circles:

```
for (let i = 0; i < 100; i++) {
  ctx.beginPath();
  ctx.fillStyle = "rgb(255 0 0 / 50%)";
  ctx.arc(
    random(canvas.width),
    random(canvas.height),
    random(50),
    0,
    2 * Math.PI,
  );
  ctx.fill();
}
```

You should get the basic idea — we are using a loop to run 100 iterations of this code, each one of which draws a circle in a random position on the page. `random(x)`, defined earlier in the code, returns a whole number between `0` and `x-1`. The amount of code needed would be the same whether we were drawing 100 circles, 1000, or 10,000. Only one number has to change.

If we weren't using a loop here, we'd have to repeat the following code for every circle we wanted to draw:

```
ctx.beginPath();
ctx.fillStyle = "rgb(255 0 0 / 50%)";
ctx.arc(
  random(canvas.width),
  random(canvas.height),
  random(50),
  0,
  2 * Math.PI,
);
ctx.fill();
```

This would get very boring and difficult to maintain.

Looping through a collection

Most of the time when you use a loop, you will have a collection of items and want to do something with every item.

One type of collection is the Array, which we met in the Arrays chapter of this course. But there are other collections in JavaScript as well, including Set and Map.

The for...of loop

The basic tool for looping through a collection is the for...of loop:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

for (const cat of cats) {
  console.log(cat);
}
```

In this example, `for (const cat of cats)` says:

1. Given the collection `cats`, get the first item in the collection.
2. Assign it to the variable `cat` and then run the code between the curly braces `{}`.
3. Get the next item, and repeat Step 2 until you've reached the end of the collection.

map() and filter()

JavaScript also has more specialized loops for collections, and we'll mention two of them here.

You can use `map()` to do something to each item in a collection and create a new collection containing the changed items:

```
function toUpper(string) {
  return string.toUpperCase();
}

const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

const upperCats = cats.map(toUpper);

console.log(upperCats);
// [ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

Here we pass a function into cats.map(), and `map()` calls the function once for each item in the array, passing in the item. It then adds the return value from each function call to a new array, and finally returns the new array. In this case the function we provide converts the item to uppercase,

so the resulting array contains all our cats in uppercase:

```
[ "LEOPARD", "SERVAL", "JAGUAR", "TIGER", "CARACAL", "LION" ]
```

You can use filter() to test each item in a collection, and create a new collection containing only items that match:

```
function lCat(cat) {
  return cat.startsWith("L");
}

const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

const filtered = cats.filter(lCat);

console.log(filtered);
// [ "Leopard", "Lion" ]
```

This looks a lot like `map()`, except the function we pass in returns a boolean: if it returns `true`, then the item is included in the new array. Our function tests that the item starts with the letter "L", so the result is an array containing only cats whose names start with "L":

```
[ "Leopard", "Lion" ]
```

Note that `map()` and `filter()` are both often used with *function expressions*, which you will learn about in our Functions lesson. Using function expressions we could rewrite the example above to be much more compact:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

const filtered = cats.filter((cat) => cat.startsWith("L"));
console.log(filtered);
// [ "Leopard", "Lion" ]
```

The standard for loop

In the "drawing circles" example above, you don't have a collection of items to loop through: you really just want to run the same code 100 times. In a case like that, you can use the for loop. This has the following syntax:

```
for (initializer; condition; final-expression) {
  // code to run
}
```

Here we have:

1. The keyword `for`, followed by some parentheses.
2. Inside the parentheses we have three items, separated by semicolons:
   1. An **initializer** — this is usually a variable set to a number, which is incremented to count the number of times the loop has run. It is also sometimes referred to as a **counter variable**.
   2. A **condition** — this defines when the loop should stop looping. This is generally an expression featuring a comparison operator, a test to see if the exit condition has been met.
   3. A **final-expression** — this is always evaluated (or run) each time the loop has gone through a full iteration. It usually serves to increment (or in some cases decrement) the counter variable, to bring it closer to the point where the condition is no longer `true`.
3. Some curly braces that contain a block of code — this code will be run each time the loop iterates.

> **NOTE:** Aside: Loops from Scrimba*MDN learning partner* provides a useful interactive breakdown of the `for` loop syntax.

Calculating squares

Let's look at a real example so we can visualize what these do more clearly.

```
<button id="calculate">Calculate</button>
<button id="clear">Clear</button>
<pre id="results"></pre>
```

```
const results = document.querySelector("#results");

function calculate() {
  for (let i = 1; i < 10; i++) {
    const newResult = `${i} x ${i} = ${i * i}`;
    results.textContent += `${newResult}\n`;
  }
```

```
  results.textContent += "\nFinished!\n\n";
}

const calculateBtn = document.querySelector("#calculate");
const clearBtn = document.querySelector("#clear");

calculateBtn.addEventListener("click", calculate);
clearBtn.addEventListener("click", () => (results.textContent = ""));
```

This gives us the following output:

This code calculates squares for the numbers from 1 to 9, and writes out the result. The core of the code is the `for` loop that performs the calculation.

Let's break down the `for (let i = 1; i < 10; i++)` line into its three pieces:

1. `let i = 1`: the counter variable, `i`, starts at `1`. Note that we have to use `let` for the counter, because we're incrementing it with `i++` (which is a reassignment) each time we go round the loop.
2. `i < 10`: keep going round the loop for as long as `i` is smaller than `10`.
3. `i++`: add one to `i` each time round the loop.

Inside the loop, we calculate the square of the current value of `i`, that is: `i * i`. We create a string expressing the calculation we made and the result, and add this string to the output text. We also add `\n`, so the next string we add will begin on a new line. So:

1. During the first run, `i = 1`, so we will add `1 x 1 = 1`.
2. During the second run, `i = 2`, so we will add `2 x 2 = 4`.
3. And so on…
4. When `i` becomes equal to `10` we will stop running the loop and move straight to the next bit of code below the loop, printing out the `Finished!` message on a new line.

Looping through collections with a for loop

You can use a `for` loop to iterate through a collection, instead of a `for...of` loop.

Let's look again at our `for...of` example above:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

for (const cat of cats) {
  console.log(cat);
}
```

We could rewrite that code like this:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];

for (let i = 0; i < cats.length; i++) {
  console.log(cats[i]);
}
```

In this loop we're starting `i` at `0`, and stopping when `i` reaches the length of the array. Then inside the loop, we're using `i` to access each item in the array in turn.

This works just fine, and in early versions of JavaScript, `for...of` didn't exist, so this was the standard way to iterate through an array. However, it offers more chances to introduce bugs into your code. For example:

- you might start `i` at `1`, forgetting that the first array index is zero, not 1.
- you might stop at `i <= cats.length`, forgetting that the last array index is at `length - 1`.

For reasons like this, it's usually best to use `for...of` if you can.

Sometimes you still need to use a `for` loop to iterate through an array. For example, in the code below we want to print a message listing our cats to a p element:

```
<p></p>
```

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];
const pElem = document.querySelector("p");

let myFavoriteCats = "My favorite big cats are ";

for (const cat of cats) {
  myFavoriteCats += `${cat}, `;
}

pElem.textContent = myFavoriteCats;
```

The final output sentence isn't very well-formed:

We'd prefer a grammatically correct sentence, which means handling the last cat differently:

```
My favorite big cats are Leopard, Serval, Jaguar, Tiger, Caracal, and Lion.
```

To do this we need to know when we are on the final loop iteration, and to do that we can use a `for` loop and examine the value of `i`:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];
const pElem = document.querySelector("p");

let myFavoriteCats = "My favorite big cats are ";

for (let i = 0; i < cats.length; i++) {
  if (i === cats.length - 1) {
    // We are at the end of the array
    myFavoriteCats += `and ${cats[i]}.`;
  } else {
    myFavoriteCats += `${cats[i]}, `;
  }
}

pElem.textContent = myFavoriteCats;
```

This gives us the output we want:

Exiting loops with break

If you want to exit a loop before all the iterations have been completed, you can use the break statement. We already met this in the previous article when we looked at switch statements — when a case is met in a switch statement that matches the input expression, the `break` statement immediately exits the switch statement and moves on to the code after it.

It's the same with loops — a `break` statement will immediately exit the loop and make the browser move on to any code that follows it.

Say we wanted to search through an array of contacts and telephone numbers and return just the number we wanted to find? First, some simple HTML — a text input allowing us to enter a name to search for, a button element to submit a search, and a p element to display the results in:

```
<label for="search">Search by contact name: </label>
<input id="search" type="text" />
<button>Search</button>
```

```
<p></p>
```

Now on to the JavaScript:

```
const contacts = [
  "Chris:2232322",
  "Sarah:3453456",
  "Bill:7654322",
  "Mary:9998769",
  "Dianne:9384975",
];
const para = document.querySelector("p");
const input = document.querySelector("input");
const btn = document.querySelector("button");

btn.addEventListener("click", () => {
  const searchName = input.value.toLowerCase();
  input.value = "";
  input.focus();
  para.textContent = "";
  for (const contact of contacts) {
    const splitContact = contact.split(":");
    if (splitContact[0].toLowerCase() === searchName) {
      para.textContent = `${splitContact[0]}'s number is ${splitContact[1]}.`;
      break;
    }
  }
  if (para.textContent === "") {
    para.textContent = "Contact not found.";
  }
});
```

1. First of all, we have some variable definitions — we have an array of contact information, with each item being a string containing a name and phone number separated by a colon.
2. Next, we attach an event listener to the button (`btn`) so that when it is pressed some code is run to perform the search and return the results.
3. We store the value entered into the text input in a variable called `searchName`, before then emptying the text input and focusing it again, ready for the next search. Note that we also run the `toLowerCase()` method on the string, so that searches will be case-insensitive.
4. Now on to the interesting part, the `for...of` loop:

1. Inside the loop, we first split the current contact at the colon character, and store the resulting two values in an array called `splitContact`.
2. We then use a conditional statement to test whether `splitContact[0]` (the contact's name, again lower-cased with `toLowerCase()`) is equal to the inputted `searchName`. If it is, we enter a string into the paragraph to report what the contact's number is, and use `break` to end the loop.

5. After the loop, we check whether we set a contact, and if not we set the paragraph text to "Contact not found.".

Skipping iterations with continue

The continue statement works similarly to `break`, but instead of breaking out of the loop entirely, it skips to the next iteration of the loop. Let's look at another example that takes a number as an input, and returns only the numbers that are squares of integers (whole numbers).

The HTML is basically the same as the last example — a simple numeric input, and a paragraph for output.

```
<label for="number">Enter number: </label>
<input id="number" type="number" />
<button>Generate integer squares</button>

<p>Output:</p>
```

The JavaScript is mostly the same too, although the loop itself is a bit different:

```
const para = document.querySelector("p");
const input = document.querySelector("input");
const btn = document.querySelector("button");

btn.addEventListener("click", () => {
  para.textContent = "Output: ";
  const num = input.value;
  input.value = "";
  input.focus();
  for (let i = 1; i <= num; i++) {
    let sqRoot = Math.sqrt(i);
    if (Math.floor(sqRoot) !== sqRoot) {
      continue;
    }
```

```
    para.textContent += `${i} `;
  }
});
```

Here's the output:

1. In this case, the input should be a number (`num`). The `for` loop is given a counter starting at 1 (as we are not interested in 0 in this case), an exit condition that says the loop will stop when the counter becomes bigger than the input `num`, and an iterator that adds 1 to the counter each time.
2. Inside the loop, we find the square root of each number using `Math.sqrt(i)`, then check whether the square root is an integer by testing whether it is the same as itself when it has been rounded down to the nearest integer (this is what `Math.floor()` does to the number it is passed).
3. If the square root and the rounded down square root do not equal one another (`!==`), it means that the square root is not an integer, so we are not interested in it. In such a case, we use the `continue` statement to skip on to the next loop iteration without recording the number anywhere.
4. If the square root is an integer, we skip past the `if` block entirely, so the `continue` statement is not executed; instead, we concatenate the current `i` value plus a space at the end of the paragraph content.

while and do...while

`for` is not the only type of general loop available in JavaScript. There are actually many others and, while you don't need to understand all of these now, it is worth having a look at the structure of a couple of others so that you can recognize the same features at work in a slightly different way.

First, let's have a look at the `while` loop. This loop's syntax looks like so:

```
initializer
while (condition) {
  // code to run


  final-expression
}
```

This works in a very similar way to the `for` loop, except that the initializer variable is set before the loop, and the final-expression is included inside the loop after the code to run, rather than these two items being included inside the parentheses. The condition is included inside the parentheses, which are preceded by the `while` keyword rather than `for`.

The same three items are still present, and they are still defined in the same order as they are in the for loop. This is because you must have an initializer defined before you can check whether or not the condition is true. The final-expression is then run after the code inside the loop has run (an iteration has been completed), which will only happen if the condition is still true.

Let's have a look again at our cats list example, but rewritten to use a while loop:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];
const pElem = document.querySelector("p");

let myFavoriteCats = "My favorite big cats are ";

let i = 0;

while (i < cats.length) {
  if (i === cats.length - 1) {
    myFavoriteCats += `and ${cats[i]}.`;
  } else {
    myFavoriteCats += `${cats[i]}, `;
  }

  i++;
}

pElem.textContent = myFavoriteCats;
```

This still works just the same as expected:

The `do...while` loop is very similar, but provides a variation on the while structure:

```
initializer
do {
  // code to run

  final-expression
} while (condition)
```

In this case, the initializer again comes first, before the loop starts. The keyword directly precedes the curly braces containing the code to run and the final expression.

The main difference between a `do...while` loop and a `while` loop is that *the code inside a* `do...while` *loop is always executed at least once*. That's because the condition comes after the

code inside the loop. So we always run that code, then check to see if we need to run it again. In `while` and `for` loops, the check comes first, so the code might never be executed.

Let's rewrite our cat listing example again to use a `do...while` loop:

```
const cats = ["Leopard", "Serval", "Jaguar", "Tiger", "Caracal", "Lion"];
const pElem = document.querySelector("p");

let myFavoriteCats = "My favorite big cats are ";

let i = 0;

do {
  if (i === cats.length - 1) {
    myFavoriteCats += `and ${cats[i]}.`;
  } else {
    myFavoriteCats += `${cats[i]}, `;
  }

  i++;
} while (i < cats.length);

pElem.textContent = myFavoriteCats;
```

Again, this works just the same as expected:

> **WARNING:** With any kind of loop, you must make sure that the initializer is incremented or, depending on the case, decremented, so the condition eventually becomes false. If not, the loop will go on forever, and either the browser will force it to stop, or it will crash. This is called an **infinite loop**.

Implementing a launch countdown

In this exercise, we want you to print out a simple launch countdown to the output box, from 10 down to Blastoff.

To complete the exercise:

1. Click **"Play"** in the code block below to edit the example in the MDN Playground.

2. Add code to loop from 10 down to 0. We've provided you with an initializer — `let i = 10;`.
3. For each iteration, create a new paragraph and append it to the output `<div>`, which we've selected using `const output = document.querySelector('.output');`. We've provided you with three code lines inside comments that need to be used somewhere inside the loop:
   1. `const para = document.createElement('p');` — creates a new paragraph.
   2. `output.appendChild(para);` — appends the paragraph to the output `<div>`.
   3. `para.textContent =` — makes the text inside the paragraph equal to whatever you put on the right-hand side, after the equals sign.
4. For the different iteration numbers listed below, write code to insert the required text inside the paragraph (you'll need a conditional statement and multiple `para.textContent =` lines):
   1. If the number is 10, print "Countdown 10" to the paragraph.
   2. If the number is 0, print "Blast off!" to the paragraph.
   3. For any other number, print just the number to the paragraph.
5. Remember to include an iterator! However, in this example we are counting down after each iteration, not up, so you **don't** want `i++` — how do you iterate downwards?

> **NOTE:** If you start typing the loop (for example `(while(i>=0)`), the browser might get stuck in an infinite loop because you have not yet entered the end condition. So be careful with this. You can start writing your code in a comment to deal with this issue and remove the comment after you finish.

If you make a mistake, you can clear your work using the *Reset* button in the MDN Playground. If you get really stuck, you can view the solution below the live output.

```
<div class="output"></div>
```

```
html {
  font-family: sans-serif;
}

h2 {
  font-size: 16px;
}

.a11y-label {
  margin: 0;
  text-align: right;
  font-size: 0.7rem;
  width: 98%;
```

```
}

body {
  margin: 10px;
  background: #f5f9fa;
}

.output {
  height: 410px;
  overflow: auto;
}
```

```
const output = document.querySelector(".output");
output.textContent = "";

// let i = 10;

// const para = document.createElement('p');
// para.textContent = ;
// output.appendChild(para);
```

Click here to show the solution

Your finished JavaScript should look something like this:

```
const output = document.querySelector(".output");
output.textContent = "";

let i = 10;

while (i >= 0) {
  const para = document.createElement("p");
  if (i === 10) {
    para.textContent = `Countdown ${i}`;
  } else if (i === 0) {
    para.textContent = "Blast off!";
  } else {
    para.textContent = i;
```

```
  }

  output.appendChild(para);

  i--;
}
```

Filling in a guest list

In this exercise, we want you to take a list of names stored in an array and put them into a guest list. But it's not quite that easy — we don't want to let Phil and Lola in because they are greedy and rude, and always eat all the food! We have two lists, one for guests to admit, and one for guests to refuse.

To complete the exercise:

1. Click **"Play"** in the code block below to edit the example in the MDN Playground.
2. Write a loop that will iterate through the `people` array.
3. During each loop iteration, check if the current array item is equal to "Phil" or "Lola" using a conditional statement:
   1. If it is, concatenate the array item to the end of the `refused` paragraph's `textContent`, followed by a comma and a space.
   2. If it isn't, concatenate the array item to the end of the `admitted` paragraph's `textContent`, followed by a comma and a space.

We've already provided you with:

- `refused.textContent +=` — the beginnings of a line that will concatenate something to the end of `refused.textContent`.
- `admitted.textContent +=` — the beginnings of a line that will concatenate something to the end of `admitted.textContent`.

Extra bonus question — after completing the above tasks successfully, you will be left with two lists of names, separated by commas, but they will be untidy — there will be a comma at the end of each one. Can you work out how to write lines that slice the last comma off in each case, and add a full stop to the end? Have a look at the Useful string methods article for help.

If you make a mistake, you can clear your work using the *Reset* button in the MDN Playground. If you get really stuck, you can view the solution below the live output.

```
<div class="output">
  <p class="admitted">Admit:</p>
  <p class="refused">Refuse:</p>
</div>
```

```
html {
  font-family: sans-serif;
}

h2 {
  font-size: 16px;
}

.a11y-label {
  margin: 0;
  text-align: right;
  font-size: 0.7rem;
  width: 98%;
}

body {
  margin: 10px;
  background: #f5f9fa;
}

.output {
  height: 100px;
  overflow: auto;
}
```

```
const people = [
  "Chris",
  "Anne",
  "Colin",
  "Terri",
  "Phil",
  "Lola",
  "Sam",
  "Kay",
```

```
  "Bruce",
];

const admitted = document.querySelector(".admitted");
const refused = document.querySelector(".refused");
admitted.textContent = "Admit: ";
refused.textContent = "Refuse: ";

// loop starts here

// refused.textContent += ...;
// admitted.textContent += ...;
```

Click here to show the solution

Your finished JavaScript should look something like this:

```
const people = [
  "Chris",
  "Anne",
  "Colin",
  "Terri",
  "Phil",
  "Lola",
  "Sam",
  "Kay",
  "Bruce",
];

const admitted = document.querySelector(".admitted");
const refused = document.querySelector(".refused");

admitted.textContent = "Admit: ";
refused.textContent = "Refuse: ";

for (const person of people) {
  if (person === "Phil" || person === "Lola") {
    refused.textContent += `${person}, `;
```

```
    } else {
      admitted.textContent += `${person}, `;
    }
  }

  refused.textContent = `${refused.textContent.slice(0, -2)}.`;
  admitted.textContent = `${admitted.textContent.slice(0, -2)}.`;
```

Which loop type should you use?

If you're iterating through an array or some other object that supports it, and don't need access to the index position of each item, then `for...of` is the best choice. It's easier to read and there's less to go wrong.

For other uses, `for`, `while`, and `do...while` loops are largely interchangeable. They can all be used to solve the same problems, and which one you use will largely depend on your personal preference — which one you find easiest to remember or most intuitive. We would recommend `for`, at least to begin with, as it is probably the easiest for remembering everything — the initializer, condition, and final-expression all have to go neatly into the parentheses, so it is easy to see where they are and check that you aren't missing them.

Let's have a look at them all again.

First `for...of`:

```
for (const item of array) {
  // code to run
}
```

`for`:

```
for (initializer; condition; final-expression) {
  // code to run
}
```

`while`:

```
initializer
while (condition) {
```

```
  // code to run

  final-expression
}
```

and finally `do...while`:

```
initializer
do {
  // code to run

  final-expression
} while (condition)
```

> **NOTE:** There are other loop types/features too, which are useful in advanced/specialized situations and beyond the scope of this article. If you want to go further with your loop learning, read our advanced Loops and iteration guide.

Summary

This article has revealed to you the basic concepts behind, and different options available when looping code in JavaScript. You should now be clear on why loops are a good mechanism for dealing with repetitive code and raring to use them in your own examples!

In the next article, we'll give you some tests that you can use to check how well you've understood and retained this information.

See also

- Loops and iteration in detail
- for...of reference
- for statement reference
- while and do...while references
- break and continue references

□□□ #3
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ cocobook-sync □□□ 2026-11-04 17:32:42 UTC