

toString vs String difference



```
const oldArray = [10, 20, 30];

const newArray = oldArray?.map((item) => item?.toString());

console.log("oldArray:", oldArray);
console.log("newArray:", newArray);
```

here we are using `toString()`

and if we use `String`

```
const newArray = oldArray?.map((item) => String(item));
```

both converts number to string what is the difference between them. Explain

Here is the **clear and beginner-friendly explanation** of the difference between `toString()` and `String()`:

✓ Difference Between `toString()` and `String()`

Both convert data to `string`, but they work **differently** and behave **differently in edge cases**.

◆ 1. `toString() → Method of the value`

`toString()` is a **method** that belongs to the actual value.

Example:

- Number has `toString()`
- Boolean has `toString()`
- Array has `toString()`
- Object has `toString()`
- String has `toString()`

✓ Works like:

```
js

item.toString()
```

✗ But it fails if the value is `null` or `undefined`.

Example:

```
js
let a = null;
a.toString(); // ✗ TypeError: Cannot read properties of null
```

◆ 2. `String()` → Global function

`String()` is a **global function**, not a method.

It converts **anything** to string safely.

✓ Works like:

```
js
String(item)
```

✓ Works on `null` and `undefined` also:

```
js
String(null) // "null"
String(undefined) // "undefined"
```

So it is **safer** compared to `toString()`.

🔍 Why do both convert number to string in your array example?

Because your array contains only numbers, and numbers have a valid `toString()` method.

```
js
[10, 20, 30].map(item => item.toString()); // ["10", "20", "30"]
[10, 20, 30].map(item => String(item)); // ["10", "20", "30"]
```

So output is **same**.

🔥 Main Differences Summary

Feature	<code>toString()</code>	<code>String()</code>
Type	Method	Function
Calls on value directly?	Yes, <code>item.toString()</code>	No, <code>String(item)</code>
Works on <code>null/undefined</code>	✗ No (throws error)	✓ Yes

Feature	<code>toString()</code>	<code>String()</code>
Safe to use everywhere	✗ No	✓ Yes
Converts value to string	✓ Yes	✓ Yes

📌 When should you use what?

✓ Use `String()` when value may be:

- `null`
- `undefined`
- or mixed types

✓ Use `toString()` when you are sure the value is not null/undefined and you want the type's own formatting.

Example:

```
js
(10).toString(2) // "1010" (binary)
```

`String(10, 2)` ✗ won't work like that.

⭐ Final Conclusion

In normal use cases, **both convert a number to a string**, but:

- **String() is safer**
- **toString() can throw errors**

If you want, I can also explain how they work internally in JavaScript engine.



we are writing it `.toString()` wher it present at global object or at prototype of anything and `String()` where it present fero whgere we are getting it

Great question, Vipul!

Let's break it down **clearly and slowly** so you understand where each one actually lives.

✓ 1. Where does `.toString()` come from?

`.toString()` is NOT from the global object.

It comes from the **prototype** of the specific data type.

Every JavaScript value (except **null** and **undefined**) inherits a **toString()** method from its prototype.

Examples:

✓ For numbers:

```
javascript
Number.prototype.toString
```

✓ For strings:

```
javascript
String.prototype.toString
```

✓ For arrays:

```
javascript
Array.prototype.toString
```

✓ For objects:

```
javascript
Object.prototype.toString
```

- ◆ This means when you write:

```
js
item.toString()
```

JavaScript does this internally:

1. Check if **item** has **toString** method → not found
2. Go to **item**'s prototype → found
3. Call it

✗ Special case

If **item** is **null** or **undefined**, there is no prototype ⇒

```
js
null.toString() // ✗ error
undefined.toString() // ✗ error
```

Because:

- `null` has NO prototype
- `undefined` has NO prototype

★ So `.toString()` lives inside the prototype chain, not globally.

✓ 2. Where does `String()` come from?

`String()` is a **built-in global function**.

It lives on the **global object** (`window` in browser, `global` in Node.js) as a **constructor function**.

It is the same place where we have:

- `Number()`
- `Boolean()`
- `Array()`
- `Object()`

So:

```
javascript
String()
↑
This is a global function (constructor)
```

When you call:

```
js
String(item)
```

JS internally does:

1. Check global scope → finds `String` function
2. Runs it → converts item to string safely

`String()` also internally handles `null` and `undefined`, so:

```
js
String(null) → "null"
```

```
String(undefined) → "undefined"
```

No error.

🔥 Summary Table

Feature	.toString()	String()
Where is it defined?	On prototype of the value (e.g., Number.prototype)	Global object (String constructor)
How is it called?	value.toString()	String(value)
Works on null/undefined?	✗ No	✓ Yes
Throws error?	If value is null/undefined	Never

✓ Final Simple Explanation

toString()

- 👉 present in **prototype** of Number, Boolean, Array, Object, etc.
- 👉 must be called on a value.
- 👉 fails on null/undefined.

String()

- 👉 present in **global scope** (like Number(), Boolean()).
- 👉 does NOT depend on value prototype.
- 👉 safely converts anything.

If you want, I can show you proof using code like:

```
js

console.log(Object.getPrototypeOf(10))
console.log(String)
```

Just tell me!



yes

Here is the **complete proof with code**, written in a very clear and simple way so you understand exactly where **toString()** and **String()** come from.

You can copy-paste all this into Chrome console or VS Code.

✓ 1. Proof: `.toString()` comes from prototype

✓ For number:

```
js
console.log( (10).__proto__.toString );
```

OR

```
js
console.log( Number.prototype.toString );
```

Both will print the same function → means `.toString()` is coming from **Number.prototype**.

✓ For array:

```
js
console.log( [].__proto__.toString );
console.log( Array.prototype.toString );
```

✓ For object:

```
js
console.log( ({}).__proto__.toString );
console.log( Object.prototype.toString );
```

✓ Proof that null/undefined have no prototype

```
js
console.log(Object.getPrototypeOf(null)); // null
console.log(Object.getPrototypeOf(undefined)); // error
```

That's why:

```
js
null.toString(); // ✗ error
undefined.toString(); // ✗ error
```

🔥 2. Proof: `String()` is a GLOBAL function

```
js

console.log(String);
```

You will see output like:

```
scss

function String() { [native code] }
```

This shows:

- `String()` exists globally
- It is a constructor function
- It is not part of `Number.prototype`, `Array.prototype`, etc.

🎯 3. Compare results clearly

`.toString()` on number

```
js

console.log( (10).toString() ); // "10"
```

`String()` on number

```
js

console.log( String(10) ); // "10"
```

Both work — output SAME because value is valid.

❌ But `.toString()` fails on null:

```
js

null.toString();
// TypeError: Cannot read properties of null
```

But `String()` works:

```
js

String(null); // "null"
String(undefined); // "undefined"
```

★ FINAL SUMMARY

Feature	.toString()	String()
Lives in	prototype chain (Number.prototype etc.)	global object
Type	method	constructor/function
Works on	all normal values	<i>all values including null</i>
Null/undefined	✗ throws error	✓ returns "null" or "undefined"
Why?	Because null/undefined have no prototype	Because String() handles everything internally

If you want, I can also show you **internal JavaScript algorithm** for how each one works.



if we write
 1. `toString()` or `window.toString`
 2. `window.String`
 output?

Here is the **exact output** and **why it happens** for all three cases.

I'll explain each one very clearly.

✓ Case 1: `toString()`

When you write:

```
js
toString()
```

JS will search like this:

1. Look in local scope → not found
2. Look in global object (`window`) → found `window.toString`

So you are actually calling:

```
js
window.toString()
```

✓ Output:

```
js
"function Window() { [native code] }"
```

This is the **string representation of the window object**, because inside `toString()`, JavaScript converts `this` (the object it's attached to) into a string.

✓ Case 2: `window.toString`

When you write:

```
js
window.toString
```

You are **not calling** the function, just referring to it.

✓ Output:

```
js
function toString() { [native code] }
```

This prints the actual function itself.

If you run:

```
js
window.toString()
```

Then → "function Window() { [native code] }"

✓ Case 3: `window.String`

When you write:

```
js
window.String
```

JS prints the global **String constructor function**.

✓ Output:

```
scss
function String() { [native code] }
```

Because **String** is a constructor like **Number**, **Boolean**, etc.

🎯 FINAL SUMMARY TABLE

Code	What it is	Output
<code>toString()</code>	Calls <code>window.toString()</code>	<code>"function Window() { [native code] }"</code>
<code>window.toString</code>	Reference to function	<code>function toString() { [native code] }</code>
<code>window.String</code>	Global String constructor	<code>function String() { [native code] }</code>

⭐ Bonus Tip

If you run:

```
js
String()
```

Output:

```
arduino
""
```

Because calling `String()` with no argument returns empty string.

If you want, I can show where these come from in the browser's global object visually.



when it is not present at global object than how and from where we are getting it from window object
`window.toString()`

Great question!

This is the **exact point where JavaScript beginners get confused**.

Let me explain it **very clearly**, step-by-step, in simple language.

✓ Why does `window.toString()` exist, even though we didn't define it?

Because **every object in JavaScript automatically inherits methods from its prototype chain**.

`window` is also an object.

And **all objects in JavaScript inherit from `Object.prototype`**.

⭐ Key Point:

`toString()` is **not** on the global object.

`toString()` comes from:

```
javascript  
Object.prototype.toString
```

The `window` object **inherits** this method because:

```
javascript  
window → Window.prototype → EventTarget.prototype → Object.prototype
```

So the chain ends at:

```
javascript  
Object.prototype.toString
```

🔥 Therefore:

When you call:

```
js  
window.toString()
```

JavaScript does this lookup:

1. Check if `window` has its own `toString` → ✗ not found
2. Check `Window.prototype` → ✗ not found
3. Check `EventTarget.prototype` → ✗ not found
4. Check `Object.prototype` → ✓ FOUND!
5. Use `Object.prototype.toString`

That is where it comes from.

🧠 This means:

`toString()` is NOT a global function.

It is a method that **all objects automatically get** because of how JavaScript's prototype system works.

🧪 PROOF (copy-paste this into console)

1. Check if `window` has its own `toString`:

```
js

window.hasOwnProperty("toString") // false
```

2. Check if Object.prototype has it:

```
js

Object.prototype.hasOwnProperty("toString") // true
```

✓ This proves `window.toString()` is coming from `Object.prototype`.

Final Explanation (Simple)

`toString()` is not global.

It is not defined in `window`.

It is inherited by `window` from `Object.prototype`.

That's why this works:

```
js

window.toString()
```

Because `window` → is an object
And all objects → inherit `toString()`.

Final Summary

Question	Answer
Is <code>toString()</code> global?	✗ No
Is <code>toString()</code> inside <code>window</code> ?	✗ Not directly
Then why does <code>window.toString()</code> work?	✓ Because <code>window</code> inherits it from <code>Object.prototype</code>
Where does <code>toString()</code> actually live?	★ <code>Object.prototype.toString</code>

If you want, I can show you the **full prototype chain of `window`**.



EventTarget.prototype what is it ?

Here is the **simplest and clearest possible explanation** of what `EventTarget.prototype` is — with ZERO confusion.

✓ What is EventTarget?

EventTarget is a **built-in browser interface** (a class) that all objects use when they need to **listen for events**.

Examples of things that inherit from EventTarget:

- `window`
- `document`
- `HTMLElement` (div, button, input...)
- `XMLHttpRequest`
- `WebSocket`
- `AbortController`

All of these can do:

- `.addEventListener()`
- `.removeEventListener()`
- `.dispatchEvent()`

These methods come from **EventTarget.prototype**.

🎯 Simple definition:

EventTarget is the "parent class" that gives event-related methods to browsers objects.

EventTarget.prototype contains:

- `addEventListener`
- `removeEventListener`
- `dispatchEvent`

✓ Prototype chain using window (very important)

When you write:

```
js
window.addEventListener(...)
```

Where does this method come from?

Prototype chain for `window`:

```
javascript

window
^
Window.prototype
^
EventTarget.prototype    <-- addEventListener is here
^
Object.prototype        <-- toString is here
```

Proof: print the prototype chain

```
js

console.log(Object.getPrototypeOf(window));
// Window.prototype
```

```
js

console.log(Object.getPrototypeOf(Object.getPrototypeOf(window)));
// EventTarget.prototype
```

```
js

console.log(Object.getPrototypeOf(EventTarget.prototype));
// Object.prototype
```

Why we discussed EventTarget?

Because earlier you asked:

Why does `window.toString()` work if `toString` is not in global?

Answer:

- `window` is an object
- It inherits through this chain:
`window → Window.prototype → EventTarget.prototype → Object.prototype`
- `toString()` lives at the **bottom**: `Object.prototype.toString`

So `window.toString()` works.

Short Layman Explanation

Think of JavaScript like a family:

- **Object.prototype** → The grandfather
- **EventTarget.prototype** → The father who gives **event features**

- **Window.prototype** → A child who inherits from EventTarget
- **window** → The actual object you use

That's why:

- **window.toString()** → from grandfather (Object.prototype)
- **window.addEventListener()** → from father (EventTarget.prototype)

⭐ Final Summary

Prototype	Purpose
EventTarget.prototype	Provides event methods like addEventListener
Inherited by	window, document, DOM elements, etc.
Why important?	It sits in the prototype chain between Window and Object

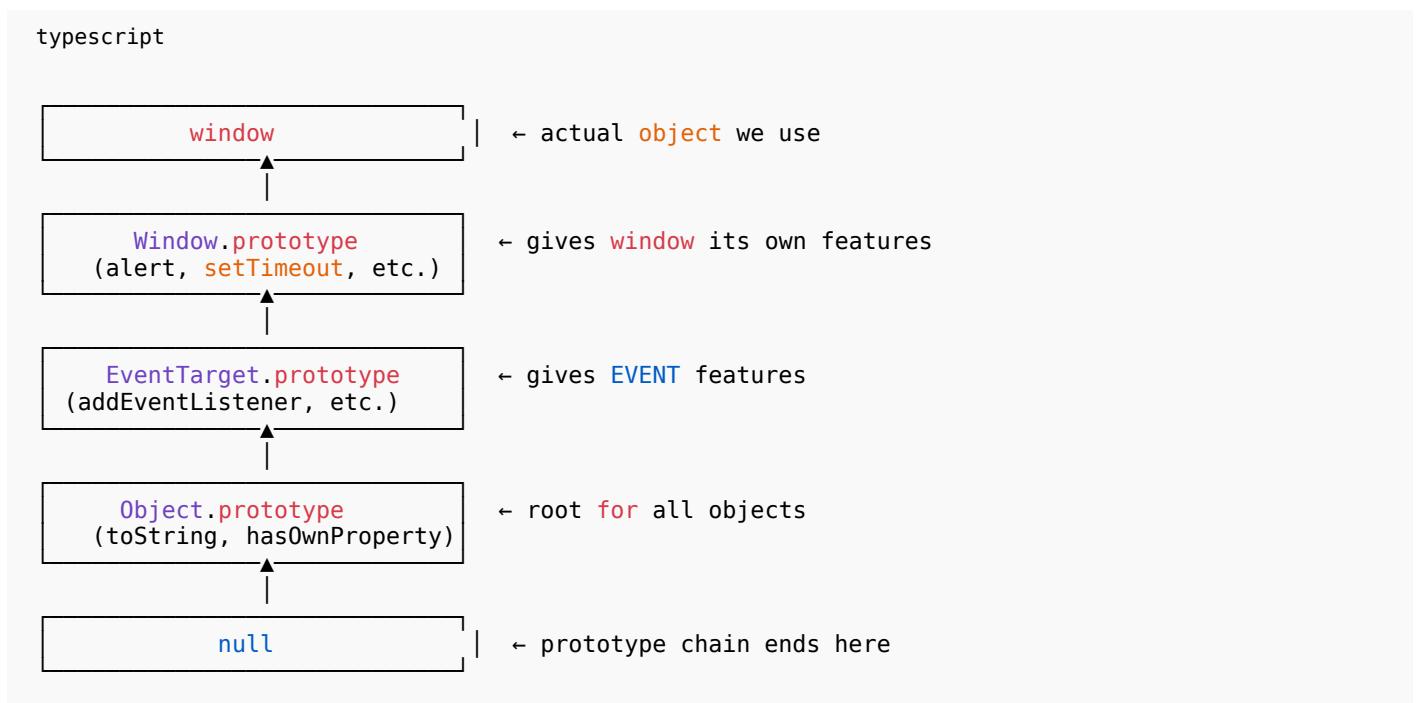
If you want, I can draw the **full prototype chain diagram** for **window**, **document**, or any DOM element.



yes draw diag

Here is the **full prototype chain diagram** of **window**, drawn in a **clear and beginner-friendly** way so you understand exactly how inheritance works.

⭐ Prototype Chain Diagram for window



🔥 Key Understanding

1 window

The object you interact with.

2 Window.prototype

Defines window-specific functions:

- `alert()`
- `confirm()`
- `setTimeout()`
- `setInterval()`
- and many more...

3 EventTarget.prototype

Defines event-related functions:

- `addEventListener()`
- `removeEventListener()`
- `dispatchEvent()`

4 Object.prototype

Defines basic object methods:

- `toString()`
- `valueOf()`
- `hasOwnProperty()`

5 null

Every prototype chain ends at `null`.

🧠 Why you see `window.toString()` working

Because JavaScript looks upward through the chain:

```
javascript
window → Window.prototype → EventTarget.prototype → Object.prototype
```

It finds `toString()` at `Object.prototype`.

💡 Want diagrams for document or DOM elements (like div, button)?

I can draw this too:

css

```
div → HTMLDivElement → HTMLElement → Element → Node → EventTarget → Object → null
```

Just tell me "**draw div prototype chain**" and I'll make the diagram.