

Reference data types JS

Reference data type in JavaScript

In JavaScript, **reference data types** are types where variables hold references (or pointers) to the actual data stored in memory, rather than holding the data directly. This contrasts with **primitive data types**, where the variable contains the actual value.

Reference Data Types in JavaScript

The main reference data types are:

1. Objects

Objects are collections of key-value pairs. They can represent more complex data structures.

```
js

const obj = { name: 'Alice', age: 25 };
```

2. Arrays

Arrays are ordered lists of values (which themselves can be primitives or references).

```
js

const arr = [1, 2, 3, 4];
```

3. Functions

Functions in JavaScript are also objects and can be treated as first-class citizens.

```
js

function greet() {
  console.log('Hello!');
}
```

4. Other built-in objects

Examples include Date, RegExp, Map, Set, WeakMap, WeakSet, etc.

How reference types work

- When you assign or pass a reference type, you are copying the reference to the object, **not the actual object itself**.

js

```
const a = { value: 10 };  
const b = a;  
b.value = 20;  
console.log(a.value); // 20, because a and b reference the same object
```

- This means that modifying the object via one variable will reflect when accessed through another variable pointing to the same object.

Summary Table

Data Type	Example	Behavior
Primitive	<code>let x = 42;</code>	Stores actual value
Reference (Object)	<code>let obj = {a:1};</code>	Stores reference to object
Reference (Array)	<code>let arr = [1, 2];</code>	Stores reference to array object
Reference (Function)	<code>function foo() {}</code>	Stores reference to function

If you want, I can explain how to copy reference types properly or deep clone objects too! Want me to?

Yes

Great! Let's dive into **copying reference types** in JavaScript and the concept of **deep cloning**.

Copying Reference Types in JavaScript

When you assign an object or array to another variable, you're copying the **reference**, not the actual data.

js

```
const original = { name: 'Alice' };
const copy = original;

copy.name = 'Bob';

console.log(original.name); // Output: 'Bob' — because both point to the same object
```

How to Create Copies

1. Shallow Copy

A shallow copy creates a new object or array, but nested objects or arrays inside still reference the original.

For Objects:

- Using `Object.assign()`

js

```
const original = { name: 'Alice', address: { city: 'NY' } };
const copy = Object.assign({}, original);

copy.name = 'Bob';
copy.address.city = 'LA';

console.log(original.name); // 'Alice' (primitive, changed independently)
console.log(original.address.city); // 'LA' (nested object still shared)
```

- Using spread syntax `{ ...obj }`

js

```
const copy2 = { ...original };
```

For Arrays:

- Using `slice()` or spread syntax:

```
js

const arr = [1, 2, 3];
const arrCopy = arr.slice();
// or
const arrCopy2 = [...arr];
```

2. Deep Copy

A deep copy duplicates everything recursively, so changes in the copy don't affect the original, even for nested objects.

Simple way (with limitations):

- Using `JSON.stringify()` and `JSON.parse()`:

```
js

const original = { name: 'Alice', address: { city: 'NY' } };
const deepCopy = JSON.parse(JSON.stringify(original));

deepCopy.address.city = 'LA';

console.log(original.address.city); // 'NY' (unchanged)
```

Limitations:

- Doesn't work with functions, `undefined`, `Symbol`, `Date`, `RegExp`, or objects with circular references.

More robust way:

- Use libraries like **Lodash** (`_.cloneDeep()`):

```
js

import _ from 'lodash';

const deepCopy = _.cloneDeep(original);
```

Summary

Copy Type	Copies Nested Objects?	Use Case	Example Method
Shallow Copy	No	When nested objects don't change	<code>Object.assign(), { ...obj }</code>
Deep Copy	Yes	When full independent copy needed	<code>JSON.parse(JSON.stringify(obj))</code> , <code>_.cloneDeep()</code>

Want me to show examples of deep copying complex objects or explain cloning with circular references?

explain cloning with circular references

Ah, circular references — this is where things get tricky!

What Are Circular References?

A **circular reference** happens when an object references itself directly or indirectly. For example:

```
js
const obj = {};
obj.self = obj; // obj has a property that points back to itself
```

Trying to deep clone such an object with `JSON.stringify()` will **fail** with a `TypeError: Converting circular structure to JSON` because `JSON.stringify` can't handle cycles.

Why Does This Happen?

When you try to serialize the object to JSON, the process goes like this:

- Encounters `obj.self` → references `obj` again → goes back to `obj.self` → infinite loop!
-

How to Clone Objects with Circular References?

Approach: Use a custom deep clone function that tracks visited objects

The idea is:

- Maintain a **map** (e.g., a `WeakMap`) of already cloned objects.
 - When an object is encountered again, return the previously cloned version instead of cloning again.
 - This avoids infinite recursion.
-

Example Deep Clone with Circular Reference Support:

js

```
function deepCloneWithCircularRefs(obj, hash = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') {
    return obj; // primitives are returned as is
  }

  if (hash.has(obj)) {
    return hash.get(obj); // Return the stored clone if already visited
  }

  // Create a new object or array depending on the original
  const clone = Array.isArray(obj) ? [] : {};

  // Store this clone in the map before deep cloning properties
  hash.set(obj, clone);

  // Recursively copy properties
  for (const key in obj) {
    if (obj.hasOwnProperty(key)) {
      clone[key] = deepCloneWithCircularRefs(obj[key], hash);
    }
  }
}
```

```
    return clone;
  }
```

Example Usage:

```
js

const obj = { name: 'Alice' };
obj.self = obj;

const cloned = deepCloneWithCircularRefs(obj);

console.log(cloned.name); // Alice
console.log(cloned.self === cloned); // true — circular reference preserved in clone
console.log(cloned.self === obj); // false — cloned object is different from original
```

Summary

- Circular references cause infinite loops in naive deep cloning.
- Use a `WeakMap` to keep track of already cloned objects.
- This approach detects and preserves circular references without infinite recursion.

Would you like me to help you build a more advanced cloning function (handling special objects like Dates, Maps, or functions) or a library recommendation?

yes

Awesome! Let's build on that and create a more **robust deep clone function** that:

- Handles **circular references**
 - Copies **special object types** like `Date`, `RegExp`, `Map`, and `Set`
 - Optionally handles **functions** (usually functions are copied by reference, but we can discuss that)
-

Advanced Deep Clone Function in JavaScript

js

```
function deepCloneAdvanced(obj, hash = new WeakMap()) {  
  // Handle null or primitive types  
  if (obj === null || typeof obj !== 'object') {  
    return obj;  
  }  
  
  // Handle Date  
  if (obj instanceof Date) {  
    return new Date(obj);  
  }  
  
  // Handle RegExp  
  if (obj instanceof RegExp) {  
    return new RegExp(obj.source, obj.flags);  
  }  
  
  // Handle Map  
  if (obj instanceof Map) {  
    const result = new Map();  
    hash.set(obj, result);  
    for (const [key, value] of obj.entries()) {  
      result.set(deepCloneAdvanced(key, hash), deepCloneAdvanced(value, hash));  
    }  
    return result;  
  }  
  
  // Handle Set  
  if (obj instanceof Set) {  
    const result = new Set();  
    hash.set(obj, result);  
    for (const value of obj.values()) {  
      result.add(deepCloneAdvanced(value, hash));  
    }  
    return result;  
  }  
  
  // Handle circular references  
  if (hash.has(obj)) {  
    return hash.get(obj);  
  }  
}
```



```
}

// Handle Array or Object
const clone = Array.isArray(obj) ? [] : {};

// Save reference to handle circular refs
hash.set(obj, clone);

// Copy all own properties (including symbols)
Reflect.ownKeys(obj).forEach((key) => {
  clone[key] = deepCloneAdvanced(obj[key], hash);
});

return clone;
}
```

How to Use It

```
js

const circularObj = {
  name: 'Circle',
  meta: new Map([
    ['created', new Date()],
    ['pattern', /abc/gi]
  ])
};

circularObj.self = circularObj;

const clonedObj = deepCloneAdvanced(circularObj);

console.log(clonedObj);
console.log(clonedObj.self === clonedObj); // true
console.log(clonedObj.meta instanceof Map); // true
console.log(clonedObj.meta.get('created') instanceof Date); // true
console.log(clonedObj.meta.get('pattern') instanceof RegExp); // true
```

Notes on Functions

- Functions are copied **by reference** here (not cloned). Cloning functions with their closure/context is generally impossible and unnecessary in most cases.
 - If you want to copy functions as-is, this works fine; if you want to skip functions or replace them, you can add additional logic.
-

Want me to help you add special handling for functions or other exotic types (e.g., TypedArrays, Promise)? Or maybe show you how to integrate this into your project?