Symbols and their use cases

🧿 Symbols in JavaScript — Explained in Detail

🔷 What is a Symbol?

A **Symbol** is a **primitive data type** introduced in ES6.

It represents a **unique and immutable identifier**.

🔧 Syntax:

```
const sym = Symbol();
```

Each time you call `Symbol()`, it returns a **new unique value**.

```
const sym1 = Symbol("id");
const sym2 = Symbol("id");

console.log(sym1 === sym2); // false
```

Even though both have the same description, they are **not equal**.

🧠 Why Use Symbols?

Symbols are useful for:

- Creating unique object keys (avoids key collisions)
- Adding private or hidden properties
- Defining well-known behaviors (via **well-known symbols**)

🧰 Basic Use Case — Unique Object Keys

```
const id = Symbol("id");

const user = {
  name: "Abhi",
  [id]: 101
};

console.log(user);        // { name: "Abhi", [Symbol(id)]: 101 }
console.log(user[id]);    // 101
```

Symbols can be used as **non-enumerable** keys (not visible in loops like `for...in`).

🚫 Hidden Properties

Symbols don't show up in:

- `for...in`
- `Object.keys()`
- `JSON.stringify()`

```
for (let key in user) {
  console.log(key); // Only logs "name", not Symbol(id)
}
```

But you can still access them via:

```
Object.getOwnPropertySymbols(user); // [Symbol(id)]
```

📌 Use Case: Avoiding Name Clashes

Libraries often use Symbols to add internal properties that don't conflict with user-defined keys.

```
const internal = Symbol("internal");

class MyComponent {
  constructor() {
    this[internal] = "privateData";
  }
}
```

💡 Well-Known Symbols (Built-in Behaviors)

JavaScript has predefined symbols used to customize object behavior:

Symbol	Purpose
`Symbol.iterator`	Defines iterable behavior (used in `for...of`)
`Symbol.toPrimitive`	Custom object-to-primitive conversion
`Symbol.toStringTag`	Custom tag for `Object.prototype.toString()`
`Symbol.hasInstance`	Custom behavior for `instanceof`
`Symbol.match`, `Symbol.replace`	Custom behavior for regex methods

Example – Symbol.iterator

```
const iterable = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
  }
};

for (const val of iterable) {
  console.log(val); // 1, 2, 3
}
```

🧪 Global Symbol Registry

You can create or access **shared symbols** using `Symbol.for()`:

```
const sym1 = Symbol.for("key");
const sym2 = Symbol.for("key");

console.log(sym1 === sym2); // true
```

And retrieve the key:

```
Symbol.keyFor(sym1); // "key"
```

📌 Summary

Feature	Description
`Symbol()`	Creates a unique, immutable identifier
Object keys	Can use Symbols to avoid collisions & hide data
Well-known symbols	Enable custom behavior in built-in operations
Global registry	`Symbol.for()` shares symbols across code

Would you like examples of `Symbol.toPrimitive`, or how symbols are used in real-world libraries (like Redux or React)?