


# What Happens When You Use `this` in a Function?

## 1. With `new` keyword:

```
function Person(name) {
  this.name = name;
}

const p = new Person("Darshan");
console.log(p.name); // "Darshan"
```


 What happens:

- A new object is created.
- `this` points to that **new object**.
- Properties/methods defined using `this` are attached to that object.
- That object is returned implicitly.

## 2. Without `new` keyword:

```
function Person(name) {
  this.name = name;
}



const p = Person("Darshan"); // ✗ Missing 'new'
console.log(p);              // undefined
console.log(window.name);    // "Darshan" in non-strict mode
```

 What happens:

- `this` refers to the **global object** (`window` in browsers or `global` in Node.js) in **non-strict mode**.
- In **strict mode**, `this` becomes `undefined`, and it throws an error.
- No object is returned, and properties are accidentally attached to the global scope.

---

## Summary Table

Feature	With <code>new</code>	Without <code>new</code>
<code>this</code> refers to	The <b>new object</b> created	<b>Global object</b> or <code>undefined</code>
Returns	The new object	<code>undefined</code> (unless you return manually)
Use case	Object instantiation	Regular function call
Error-prone	No (when constructor is intended)	Yes (e.g., if using <code>this</code> for object setup)
Good practice?	 Yes	 No (unless you're not using <code>this</code> )

---

## Example to See the Difference:

```
function Car() {  
  this.make = "Toyota";  
  console.log(this);  
}  
  
Car();           // Logs: window or global object (unsafe)  
new Car();       // Logs: newly created Car object (safe)
```

---

### ☒ When to Use **new**?

Use **new** **only** when the function is intended to be a **constructor** (like a blueprint for an object).

---