this Keyword

	STRICT MODE		NON-STRICT MODE	
NODE.JS	console.log(this)	{ }	console.log(this)	{ }
	function printthis() { console.log(this) }	Undefined	function printthis() { console.log(this) }	Global Object
BROWSER	console.log(this)	Window Object	console.log(this)	Window Object
	function printthis() { console.log(this) }	Undefined	function printthis() { console.log(this) }	Window Object

1. Context-Sensitive: The value of this depends on how a function is called, not where it is defined.
2. Global Context: In the global execution context (outside any function), this refers to the global object (window in browsers).

this Keyword (Object Method)

```
const obj = {
  name: "KGCoding",
  greet: function() {
    console.log(this.name);
  }
};
obj.greet(); // Output: KGCoding
```

Object Method: When a function is called as a method of an object, this refers to the object the method is called on.

this Keyword (Constructor Function)

```
function Person(name) {
  this.name = name;
}
const person = new Person("KGCoding");
console.log(person.name); // Output: KGCoding
```

Constructor Function: When a function is used as a constructor with new, this refers to the newly created instance.

this Keyword (Event Handler)

```
<button id="myButton">Click me</button>
<script>
  document.getElementById("myButton").addEventListener("click", function() {
    console.log(this.id); // Output: myButton
  });
</script>
```

Event Handler: In an event handler, this refers to the element that received the event.

this Keyword (Arrow Functions)

```
const obj = {
  name: "KGCoding",
  greet: () => {
    console.log(this.name);
  }
};
obj.greet(); // Output: undefined (inherited from global context)
```

Arrow Functions: Arrow functions do not have their own this.
They inherit this from the enclosing object.

Class (Inheritance)

```
// Define a class that extends another class
class Dog extends Animal {
  constructor(name, breed) {
    super(name); // Call the parent class constructor
    this.breed = breed;
  }

  // Method overriding
  speak() {
    console.log(`${this.name}, the ${this.breed}, barks.`);
  }
}

// Create an instance of the subclass
const myDog = new Dog('Buddy', 'Golden Retriever');
myDog.speak(); // Output: "Buddy, the Golden Retriever, barks."
```

- Inheritance allows one class to inherit properties and methods from another class. This is achieved using the extends keyword.
- The super keyword is used to call the constructor of the parent class and to access its methods.

Class (Inheritance)

```
class MathUtils {
  // Static method
  static add(a, b) {
    return a + b;
  }
}

// Call the static method
console.log(MathUtils.add(5, 3)); // Output: 8
```

- Static methods are defined on the class itself, rather than on instances of the class. They can be used to create utility functions related to the class.
- Static methods are called on the class directly, without needing to create an instance.

Error Handling (Try-Catch Statements)

```
try {
  const data = JSON.parse('Invalid JSON');
} catch (error) {
  console.error('Failed to parse JSON:', error.message);
}
```

- The try...catch block is used to handle exceptions in synchronous code.
- The try block contains code that might throw an error, while the catch block handles the error.

Error Handling (Error Objects)

```
const error = new Error('Something went wrong');
console.log(error.name); // "Error"
console.log(error.message); // "Something went wrong"
```

- JavaScript provides the Error object for creating and handling errors.
- The Error object includes properties like name and message to describe the error.

Error Handling (Throwing Errors)

```
function divide(a, b) {
  if (b === 0) {
    throw new Error('Division by zero is not allowed');
  }
  return a / b;
}

try {
  console.log(divide(4, 0));
} catch (error) {
  console.error(error.message); // "Division by zero is not allowed"
}
```

- You can use the throw statement to throw an error manually.
- Thrown errors can be caught and handled by a try...catch block.

Error Handling (Finally Block)

```
try {
  console.log('Trying to execute');
  // Some code that may throw an error
} catch (error) {
  console.error('Caught an error:', error.message);
} finally {
  console.log('This will always execute');
}
```

- The finally block is used to execute code regardless of whether an error occurred or not.
- It is typically used for cleanup tasks like closing connections or releasing resources.

Error Handling (Extending Errors)

```
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}

function validate(input) {
  if (!input) {
    throw new ValidationError('Input is required');
  }
}

try {
  validate('');
} catch (error) {
  if (error instanceof ValidationError) {
    console.error('Validation Error:', error.message);
  }
}
```

- You can create custom error types by extending the built-in Error class.
- This is useful for defining specific error types in your application.