In JavaScript, there are several ways to create functions, each with its own syntax and use cases. Below, I'll outline the primary methods for defining functions, explain their differences, and provide examples. I'll keep it concise while covering all the key approaches.

Ways to Create Functions in JavaScript

1. **Function Declaration**
   - A function defined with the `function` keyword, hoisted to the top of its scope, so it can be called before its definition.
   - **Syntax**:

```javascript
function functionName(param1, param2) {
  return param1 + param2;
}
```

   - **Example**:

```javascript
console.log(add(2, 3)); // Works due to hoisting
function add(a, b) {
  return a + b;
}
// Output: 5
```

   - **Key Features**: Hoisted, can be named, ideal for reusable named functions.

2 Function Expression

- A function assigned to a variable, not hoisted (only the variable declaration is hoisted, if using `var`).
- Can be named or anonymous.
- **Syntax**:

```javascript
const functionName = function (param1, param2) {
  return param1 + param2;
};
```

- **Example**:

```javascript
const multiply = function (a, b) {
  return a * b;
};
console.log(multiply(4, 5)); // Output: 20
// console.log(multiply(4, 5)); // Error if called before definition
```

- **Key Features**: Not hoisted, useful for passing functions as arguments or assigning to variables.

3 Arrow Function (ES6)

- A concise syntax introduced in ES6, using `=>`. Does not have its own `this`, `arguments`, or `super`.
- **Syntax**:

```javascript
const functionName = (param1, param2) => {
  return param1 + param2;
};
// Shorter form for single expression:
const functionName = (param1, param2) => param1 + param2;
```

- **Example**:

```javascript
const subtract = (a, b) => a - b;
console.log(subtract(10, 4)); // Output: 6
```

- **Key Features**: Concise, no own `this` (inherits from surrounding scope), not hoisted, great for callbacks and one-liners.

4 Immediately Invoked Function Expression (IIFE)

- A function expression that is defined and executed immediately. Often used to create a private scope.
- **Syntax**:

```javascript
(function () {
  // Code here
})();
```

- **Example**:

```javascript
(function () {
  let privateVar = "I'm private";
  console.log(privateVar); // Output: I'm private
})();
console.log(typeof privateVar); // Output: undefined
```

- **Key Features**: Runs once, creates isolated scope, useful for initialization or avoiding global pollution.

5 Function Constructor

- Creates a function dynamically using the `Function` constructor. Rarely used due to security and performance concerns (e.g., `eval`-like risks).
- **Syntax**:

```javascript
const functionName = new Function('param1', 'param2', 'return param1 + param2');
```

- **Example**:

```javascript
const divide = new Function('a', 'b', 'return a / b');
console.log(divide(10, 2)); // Output: 5
```

- **Key Features**: Dynamic creation, not recommended for most use cases, not hoisted.

6 Method Definition in Objects

- Functions defined as properties of an object, often using shorthand syntax (ES6).
- **Syntax**:

```javascript
const obj = {
  methodName(param1) {
    return param1;
  }
};
```

- **Example**:

```javascript
const calculator = {
  add(a, b) {
    return a + b;
  }
};
console.log(calculator.add(3, 4)); // Output: 7
```

- **Key Features**: Bound to the object, uses object's `this`, concise in ES6.

7 Generator Function

- A special type of function that can pause and resume execution, yielding multiple values over time.
- **Syntax**:

```javascript
function* generatorName() {
  yield value;
}
```

- **Example**:

```javascript
function* countUp() {
  yield 1;
  yield 2;
  yield 3;
}
const counter = countUp();
console.log(counter.next().value); // Output: 1
console.log(counter.next().value); // Output: 2
console.log(counter.next().value); // Output: 3
```

- **Key Features**: Uses `yield` to pause/resume, returns an iterator, useful for async flows or iterating over sequences.

8 Async Function

- A function that handles asynchronous operations, returning a `Promise`. Defined with the `async` keyword.
- **Syntax**:

```javascript
async function functionName() {
  return await promise;
}
```

- **Example**:

```javascript
async function fetchData() {
  return "Data fetched";
}
fetchData().then(data => console.log(data)); // Output: Data fetched
```

- **Key Features**: Simplifies promise handling, uses `await`, not hoisted.

9 Async Generator Function

- Combines `async` and generator functions to yield promises over time.
- **Syntax**:

```javascript
async function* asyncGeneratorName() {
  yield await promise;
}
```

- **Example**:

```javascript
async function* fetchItems() {
  yield await Promise.resolve("Item 1");
  yield await Promise.resolve("Item 2");
}
(async () => {
  for await (const item of fetchItems()) {
    console.log(item);
  }
})();
// Output: Item 1, Item 2
```

- **Key Features**: Yields promises, combines async and generator capabilities, niche use case.

Summary of Function Types

	Type	Hoisted?	Own `this`?	Use Case
Function Declaration	Yes	Yes		General-purpose, reusable functions
Function Expression	No	Yes		Assign to variables, pass as arguments
Arrow Function	No	No (inherits)		Callbacks, concise syntax, no `this`
IIFE	No	Yes		One-time execution, private scope
Function Constructor	No	Yes		Dynamic function creation (rare)
Object Method	No	Yes (object's)		Methods in objects/classes
Generator Function	Yes (if declared)	Yes		Iterators, pause/resume execution
Async Function	No	Yes		Asynchronous operations, promises
Async Generator Function	No	Yes		Async iteration over sequences

Additional Notes

- **Hoisting**: Only function declarations (and `var` variables) are fully hoisted. Function expressions, arrow functions, and others are not.
- **Scope Interaction**: Functions create their own scope (function scope). Variables defined inside are inaccessible outside unless returned or assigned to an outer scope.
- **Call Stack**: Every function call adds an execution context to the call stack, regardless of how the function is defined. For example, calling an arrow function or a generator function still pushes it onto the stack.
- **Rest and Default Parameters**: All function types (except `Function` constructor) support rest parameters (`...args`) and default parameters (`param = value`).

Example Combining Multiple Types

javascript Collapse Unwrap Run Copy

```javascript
// Function Declaration
function start() {
  console.log("Starting...");
  return multiply(2, 3);
}

// Function Expression
const multiply = function (a, b = 1) {
  return a * b;
};

// Arrow Function
const log = msg => console.log(msg);

// IIFE
(function () {
  log("IIFE running");
})();

// Object Method
const app = {
  run() {
    log(start()); // Calls function declaration
  }
};

app.run();
// Output:
// IIFE running
// Starting...
// 6
```

Quick Quiz

1. Which function types are hoisted?
2. Why might you choose an arrow function over a function expression?
3. What's the output of `const x = () => 42; console.log(x());`?

Answers:

1. Only **function declarations** (and generator function declarations) are hoisted.
2. Arrow functions are concise, don't have their own `this`, and are ideal for callbacks or functional programming.
3. `42` (the arrow function returns `42`).

If you'd like a deeper dive into any specific function type, how they interact with scope/call stack, or examples with microtask/task queues, let me know!