Here are **conceptual examples** of JavaScript functions to demonstrate their types, properties, and uses:

---

1. Basic Function Declaration

- A simple function to add two numbers.

```
function add(a, b) {
  return a + b;
}
console.log(add(5, 3)); // Output: 8
```

---

2. Function Expressions

- Assign a function to a variable.

```
const multiply = function (a, b) {
          return a * b;
};
console.log(multiply(4, 2)); // Output: 8
```

---

3. Arrow Functions

- Concise syntax for writing functions.

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

---

4. Anonymous Functions

- A function without a name, often used in callbacks.

```
 setTimeout(function () {
  console.log("This runs after 2 seconds!");
}, 2000);
```

---

5. Immediately Invoked Function Expressions (IIFE)

- Functions that are executed immediately after definition.

```
(function () {
  console.log("IIFE executed!");
})(); // Output: "IIFE executed!"
```

---

6. Default Parameters

- Specify default values for parameters.

```
function greet(name = "Guest") {
  return `Hello, ${name}!`;
}
console.log(greet()); // Output: "Hello, Guest!"
console.log(greet("Alice")); // Output: "Hello, Alice!"
```

---

7. Rest Parameters

- Collect arguments into an array.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3, 4)); // Output: 10
```

---

8. Spread Operator in Functions

- Pass elements of an array as individual arguments.

```
const numbers = [10, 20, 30];
 console.log(Math.max(...numbers)); // Output: 30
```

---

9. Function Returning Another Function

- Higher-order functions.

```
function makeMultiplier(multiplier) {
  return function (num) {
    return num * multiplier;
  };
}
const double = makeMultiplier(2);
console.log(double(5)); // Output: 10
```

---

10. Callback Functions

- Pass a function as an argument.

```
function processArray(arr, callback) {
  for (let i = 0; i < arr.length; i++) {
    console.log(callback(arr[i]));
  }
}
processArray([1, 2, 3], (num) => num * 2); // Output: 2, 4, 6
```

---

11. Recursive Functions

- A function that calls itself.

```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
console.log(factorial(5)); // Output: 120
```

---

12. Closures

- Functions that retain access to their parent scope even after the parent has closed.

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
```

```
    return count;
  };
}
const counter = outer();
console.log(counter()); // Output: 1
console.log(counter()); // Output: 2
```

---

13. Pure Functions

- A function whose output is determined only by its input.

```
function pureAdd(a, b) {
  return a + b;
}
console.log(pureAdd(2, 3)); // Output: 5
```

---

14. Impure Functions

- A function that modifies external state.

```
let total = 0;
function impureAdd(value) {
  total += value;
}
impureAdd(5);
console.log(total); // Output: 5
```

---

15. Asynchronous Functions

- Using async and await.

```
async function fetchData() {
  const response = await fetch("https://jsonplaceholder.typicode.com/posts/1");
  const data = await response.json();
  console.log(data);
}
fetchData();
```

---

16. Generator Functions

- Special functions that can pause and resume execution.

```
function* generator() {
  yield "First";
  yield "Second";
  yield "Third";
}
const gen = generator();
console.log(gen.next().value); // Output: "First"
console.log(gen.next().value); // Output: "Second"
```

---

17. Function Hoisting

- Function declarations are hoisted to the top.

```
console.log(hoisted()); // Output: "Hoisted!"
function hoisted() {
  return "Hoisted!";
}
```

---

18. Methods Inside Objects

- Functions as object properties.

```
let person = {
  name: "Alice",
  greet() {
    return `Hello, ${this.name}!`;
  }
};
console.log(person.greet()); // Output: "Hello, Alice!"
```

---

19. Bind, Call, Apply

call()

```
function greet(city) {
  return `Hello, ${this.name} from ${city}!`;
}
const user = { name: "John" };
console.log(greet.call(user, "New York")); // Output: "Hello, John from New York!
```

apply()

```
console.log(greet.apply(user, ["Los Angeles"])); // Output: "Hello, John from Los Angeles!"
```

bind()

```
const boundGreet = greet.bind(user, "Chicago");
console.log(boundGreet()); // Output: "Hello, John from Chicago!"
```

---

20. Anonymous vs Named Functions

Named Function:

```
function square(x) {
  return x * x;
}
```

Anonymous Function:

```
const square = function (x) {
  return x * x;
};
```

Would you like detailed explanations or exercises for any of these examples?