Functions

Functions are one of the most central tools in JavaScript programming. The concept of wrapping a piece of program in a value has many uses. It gives us a way to structure larger programs, to reduce repetition, to associate names with subprograms, and to isolate these subprograms from each other.

The most obvious application of functions is defining new vocabulary. Creating new words in prose is usually bad style, but in programming, it is indispensable.

Defining a function

A function definition is a regular binding where the value of the binding is a function. For example, this code defines square to refer to a function that produces the square of a given number:

```
const square = function (x) {
  return x * x;
};
console.log(square(12)); // → 144
```

A function is created with an expression that starts with the keyword **function**. Functions have a set of ***parameters*** (in this case, only x) and a *body*, which contains the statements that are to be executed when the function is **called**. The body of a function created this way must always be wrapped in braces, even when it consists of only a single statement.

A function can have multiple parameters or no parameters at all. In the following example, makeNoise does not list any parameter names, whereas roundTo (which rounds n to the nearest multiple of step) lists two:

```
const makeNoise = function () {
  console.log("Pling!");
};

makeNoise();    // → Pling!

const roundTo = function(n, step) {
  let remainder = n % step;
  return n - remainder + (remainder < step / 2 ? 0 : step);
};
console.log(roundTo(23, 10));  // → 20
```

Some functions, such as roundTo and square, produce a value, and some don't, such as makeNoise, whose only result is a side effect. A return statement determines the value

the function returns. When control comes across such a statement, it immediately jumps out of the current function and gives the returned value to the code that **called the function**. A return keyword without an expression after it will cause the function to return undefined. Functions that don't have a return statement at all, such as makeNoise, similarly return undefined.

Parameters to a function behave like regular bindings, but their initial values are given by the *caller* of the function, not the code in the function itself.

Bindings and scopes

Each binding has a *scope*, which is the part of the program in which the binding is visible. For bindings defined outside of any function, block, or module, the scope is the whole program—you can refer to such bindings wherever you want. These are called ***global***.

Bindings created for function parameters or declared inside a function can be referenced only in that function, so they are known as ***local*** bindings. Every time the function is called, new instances of these bindings are created. This provides some isolation between functions—each function call acts in its own little world (its local environment) and can often be understood without knowing a lot about what's going on in the global environment.

Bindings declared with ***let*** and ***const*** are in fact **local** to the ***block*** in which they are declared, so if you create one of those inside of a loop, the code before and after the loop cannot "**see**" it. In pre-2015 JavaScript, only functions created new scopes, so old-style bindings, created with the **var** keyword, are visible throughout the whole function in which they appear—or throughout the **global** scope, if they are not in a function.

```
let x = 10;   // global
if (true) {
  let y = 20; // local to block
  var z = 30; // also global
}
```

Each scope can "look out" into the scope around it, so x is visible inside the block in the example. The exception is when multiple bindings have the same name—in that case, code can see only the innermost one. For example, when the code inside the halve function refers to n, it is seeing its *own* n, not the **global** n.

```
const halve = function(n) {
  return n / 2;
};
```

```
let n = 10;
console.log(halve(100)); // → 50
console.log(n);  // → 10
```

Nested scope

JavaScript distinguishes not just global and local bindings. Blocks and functions can be created inside other blocks and functions, producing multiple degrees of locality.

```
let globalVar = "I'm global"; // Global scope

function showScope() {
  let localVar = "I'm local"; // Local scope
  console.log(localVar);
}
console.log(globalVar); // Works
console.log(localVar);  // Error!
```

The set of bindings visible inside a block is determined by the place of that block in the program text. Each local scope can also see all the local scopes that contain it, and all scopes can see the global scope. This approach to binding visibility is called *lexical scoping*.

Functions as values

A function binding usually simply acts as a name for a specific piece of the program. Such a binding is defined once and never changed. This makes it easy to confuse the function and its name.

But the two are different. A function value can do all the things that other values can do—you can use it in arbitrary expressions, not just call it. It is possible to store a function value in a new binding, pass it as an argument to a function, and so on. Similarly, a binding that holds a function is still just a regular binding and can, if not constant, be assigned a new value, like so:

```
let launchMissiles = function() {
  missileSystem.launch("now");
};
if (safeMode) {
  launchMissiles = function() {/* do nothing */};
}
```

Declaration notation

There is a slightly shorter way to create a function binding. When the function keyword is used at the start of a statement, it works differently:

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

This is a function *declaration*. The statement defines the binding square and points it at the given function. It is slightly easier to write and doesn't require a semicolon after the function.

There is one subtlety with this form of function definition.

```
console.log("The future says:", future());

function future() {
  return "You'll never have flying cars";
}
```

The preceding code works, even though the function is defined ***below*** the code that uses it. Function declarations are not part of the regular top-to-bottom flow of control. They are conceptually moved to the top of their scope and can be used by all the code in that scope. This is sometimes useful because it offers the freedom to order code in a way that seems the clearest, without worrying about having to define all functions before they are used.

Arrow functions

There's a third notation for functions, which looks very different from the others. Instead of the function keyword, it uses an arrow (=>) made up of an equal sign and a greater-than character (not to be confused with the greater-than-or-equal operator, which is written >=):

```
const roundTo = (n, step) => {
  let remainder = n % step;
  return n - remainder + (remainder < step / 2 ? 0 : step);
};
```

The arrow comes *after* the list of parameters and is followed by the function's body. It expresses something like "this input (the parameters) produces this result (the body)".

When there is only one parameter name, you can **omit the parentheses** around the parameter list. If the body is a **single expression** rather than a block in braces, that expression will be returned from the function. So, these two definitions of square do the same thing:

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

When an arrow function has no parameters at all, its parameter list is just an empty set of parentheses.

```
const horn = () => {
```

```
  console.log("Toot");
};
```

There's no deep reason to have both arrow functions and function expressions in the language. Arrow functions were added in 2015, mostly to make it possible to write small function expressions in a less verbose way.

The call stack

The way control flows through functions is somewhat involved. Let's take a closer look at it. Here is a simple program that makes a few function calls:

```
function greet(who) {
  console.log("Hello " + who);
}
greet("Harry");
console.log("Bye");
```

A run through this program goes roughly like this: the call to greet causes control to jump to the start of that function (line 2). The function calls console.log, which takes control, does its job, and then returns control to line 2. There, it reaches the end of the greet function, so it returns to the place that called it—line 4. The line after that calls console.log again. After that returns, the program reaches its end.

Because a function has to jump back to the place that called it when it returns, the computer must remember the context from which the call happened. In one case, console.log has to return to the greet function when it is done. In the other case, it returns to the end of the program.

The place where the computer stores this context is the *call stack*. Every time a function is called, the current context is stored on top of this stack. When a function returns, it removes the top context from the stack and uses that context to continue execution.

Storing this stack requires space in the computer's memory. When the stack grows too big, the computer will fail with a message like "out of stack space" or "too much recursion". The following code illustrates this by asking the computer a really hard question that causes an infinite back-and-forth between two functions. Or rather, it *would* be infinite, if the computer had an infinite stack. As it is, we will run out of space, or "blow the stack".

```
function chicken() {
  return egg();
}
function egg() {
  return chicken();
}
console.log(chicken() + " came first.");   // → ??
```

Optional Arguments

The following code is allowed and executes without any problem:

```
function square(x) { return x * x; }
console.log(square(4, true, "hedgehog")); // → 16
```

We defined square with only one parameter. Yet when we call it with three, the language doesn't complain. **It ignores the extra arguments** and computes the square of the first one.

JavaScript is extremely broad-minded about the number of arguments you can pass to a function. If you pass too many, **the extra ones are ignored. If you pass too few, the missing parameters are assigned the value undefined.**

The downside of this is that it is possible—likely, even—that you'll accidentally pass the wrong number of arguments to functions. And no one will tell you about it. The upside is that you can use this behavior to allow a function to be called with different numbers of arguments. For example, this minus function tries to imitate the - operator by acting on either one or two arguments:

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));  // → -10
console.log(minus(10, 5));  // → 5
```

If you write an = operator after a parameter, followed by an expression, the value of that expression will replace the argument when it is not given. For example, this version of roundTo makes its second argument optional. If you don't provide it or pass the value undefined, it will default to one:

```
function roundTo(n, step = 1) {
  let remainder = n % step;
  return n - remainder + (remainder < step / 2 ? 0 : step);
};

console.log(roundTo(4.5)); // → 5
console.log(roundTo(4.5, 2)); // → 4
```

Anonymous Functions

An anonymous function in JavaScript is a function that does not have a name. Unlike named functions, which are declared with a specific identifier, anonymous functions are typically defined inline and are often used as arguments to other functions or assigned to variables.

Anonymous functions can be written in two ways:
1. Function Expression:

```
const myFunction = function() {
    // Function body
};
```

2. Arrow Function:

```
const myFunction = () => {
    // Function body
};
```

Examples of Anonymous Functions

1. Assigning an Anonymous Function to a **Variable**. You can assign an anonymous function to a variable and call it later.

```
const greet = function() {
   console.log("Hello, World!");
};

greet(); // Output: Hello, World!
```

2. Using Anonymous Functions as **Callbacks**. Anonymous functions are commonly used as callbacks in methods like `setTimeout`, `map`, `filter`, etc.

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

3. Passing Anonymous Functions as **Arguments**. Anonymous functions can be passed directly as arguments to other functions.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(function(num) {
   return num * 2;
});

console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

4. Arrow Function as an **Anonymous Function**. Arrow functions provide a concise way to write anonymous functions.

```
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(num => num * num);

console.log(squared); // Output: [1, 4, 9, 16, 25]
```

Advantages of Anonymous Functions
1. Conciseness: They allow you to write shorter and more readable code, especially with arrow functions.
2. No Pollution of Global Scope: Since they are not named, they don't add to the global namespace.
3. Flexibility: They can be used anywhere a function is expected, such as in callbacks or as arguments to other functions.

Disadvantages of Anonymous Functions
1. Debugging: Anonymous functions can make debugging harder because they don't have a name in stack traces.
2. Reusability: They cannot be reused since they are not named and are often defined inline.

When to Use Anonymous Functions
- When you need a function for a short, one-time use (e.g., callbacks, event handlers).
- When you want to avoid polluting the global scope.
- When using functional programming patterns like `map`, `filter`, or `reduce`.

Anonymous functions are a powerful feature in JavaScript that allow for concise and flexible code. They are widely used in modern JavaScript development, especially in functional programming and event-driven programming. However, they should be used judiciously, keeping in mind their limitations in debugging and reusability.

Recursion

It is perfectly okay for a function to call itself, as long as it doesn't do it so often that it overflows the stack. A function that calls itself is called *recursive*. Recursion allows some functions to be written in a different style.

Recursion is a programming technique where a function calls itself in order to solve a problem. It is particularly useful for tasks that can be broken down into smaller, similar subproblems. A recursive function typically has two main parts:

1. Base Case: The condition under which the recursion stops.
2. Recursive Case: The part where the function calls itself with a modified argument.

How Recursion Works

1. The function calls itself with a smaller or simpler input.
2. This process continues until the base case is reached.
3. Once the base case is reached, the function stops calling itself and starts returning values back up the call stack.

Key Concepts in Recursion

1. Base Case: Prevents infinite recursion by providing a condition to stop the recursion.
2. Recursive Case: Breaks the problem into smaller subproblems and calls the function itself.
3. Call Stack: JavaScript uses a call stack to manage function calls. Each recursive call adds a new frame to the stack until the base case is reached.

Take, for example, this power function, which does the same as the (exponentiation) operator:

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));   // → 8
```

Summary of Steps

1. **power(2, 3) calls power(2, 2).**
2. **power(2, 2) calls power(2, 1).**
3. **power(2, 1) calls power(2, 0).**

4. **power(2, 0) returns 1.**
5. **The call stack unwinds:**
    - **power(2, 1) returns 2 * 1 = 2.**
    - **power(2, 2) returns 2 * 2 = 4.**
    - **power(2, 3) returns 2 * 4 = 8.**

Thus, the function power(2, 3) correctly calculates and returns 8.

This is rather close to the way mathematicians define exponentiation and arguably describes the concept more clearly than the loop. The function calls itself multiple times with ever smaller exponents to achieve the repeated multiplication.

However, this implementation has one problem: in typical JavaScript implementations, it's about three times slower than a version using a for loop. Running through a simple loop is generally cheaper than calling a function multiple time.

Recursion is **not always just an inefficient alternative to looping**. Some problems really are easier to solve with recursion than with loops. Most often these are problems that require exploring or processing several "branches", each of which might branch out again into even more branches.

Example 1: Factorial of a Number

The factorial of a number `n` (denoted as `n!`) is the product of all positive integers less than or equal to `n`. For example: **`5! = 5 * 4 * 3 * 2 * 1 = 120`**

```
function factorial(n) {
  // Base case: factorial of 0 or 1 is 1
  if (n === 0 || n === 1) {
    return 1;
  }
  // Recursive case: n! = n * (n-1)!
  return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120
```

Step-by-Step Execution

1. `factorial(5)` calls `factorial(4)`.
2. `factorial(4)` calls `factorial(3)`.
3. `factorial(3)` calls `factorial(2)`.

4. `factorial(2)` calls `factorial(1)`.
5. `factorial(1)` hits the base case and returns `1`.
6. The call stack unwinds:
   - `factorial(2)` returns `2 * 1 = 2`.
   - `factorial(3)` returns `3 * 2 = 6`.
   - `factorial(4)` returns `4 * 6 = 24`.
   - `factorial(5)` returns `5 * 24 = 120`.

Example 2: Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. For example: `0, 1, 1, 2, 3, 5, 8, 13, ...`

```
function fibonacci(n) {
   // Base case: fibonacci(0) = 0, fibonacci(1) = 1
   if (n === 0) return 0;
   if (n === 1) return 1;
   // Recursive case: fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
   return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(6)); // Output: 8
```

Step-by-Step Execution

1. `fibonacci(6)` calls `fibonacci(5)` and `fibonacci(4)`.
2. `fibonacci(5)` calls `fibonacci(4)` and `fibonacci(3)`.
3. `fibonacci(4)` calls `fibonacci(3)` and `fibonacci(2)`.
4. This continues until the base cases (`fibonacci(0)` and `fibonacci(1)`) are reached.
5. The call stack unwinds, and the results are summed up:
   - `fibonacci(2)` returns `1`.
   - `fibonacci(3)` returns `2`.
   - `fibonacci(4)` returns `3`.
   - `fibonacci(5)` returns `5`.
   - `fibonacci(6)` returns `8`.

Advantages of Recursion

1. Simplicity: Recursive solutions are often more intuitive and easier to write for problems that can be broken into smaller subproblems.

2. Readability: Recursive code can be more readable and concise compared to iterative solutions.
3. Natural Fit: Some problems (e.g., tree traversal, backtracking) are naturally suited to recursion.

Disadvantages of Recursion

1. Performance: Recursive functions can be less efficient due to the overhead of function calls and the call stack.
2. Stack Overflow: Deep recursion can lead to a stack overflow error if the base case is not reached.
3. Debugging: Recursive code can be harder to debug due to multiple layers of function calls.