Let's learn JavaScript functions in a fun way!

--Imagine we're in a magical kingdom of CodeLand!
--Functions are like Toy Boxes!
--Think of a function like a toy box where you can store a set of instructions (toys) that can be used again and again.
--Function Syntax: The Toy Box Label!

```
function toyBoxName(instructions) {
  // toys (instructions) go here!
}
```

--Function Name: The Toy Box Title!
--Choose a name for your toy box (function). For example: buildCastle

```
function buildCastle() {
  // instructions to build a castle!
}
```

--Parameters: The Toy Box Ingredients!
--Imagine you need blocks, glue, and a hammer to build a castle. These are like parameters!

```
function buildCastle(blocks, glue, hammer) {
  // use blocks, glue, and hammer to build!
}
```

--Function Body: The Toy Box Instructions!
--Inside the toy box, write the steps to build the castle!

```
function buildCastle(blocks, glue, hammer) {
  step1: use hammer to...
  step2: apply glue to...
  step3: stack blocks...
}
```

--Calling a Function: Playtime!
--When you want to build the castle, just call the toy box (function) by its name!

```
buildCastle(100, 'strongGlue', 'mightyHammer');
```

--Return Value: The Finished Toy!
--Sometimes, the toy box returns a finished toy!

```
function buildCastle(blocks, glue, hammer) {
  // build castle...
  return finishedCastle;
}
```

Example Time!
--Let's create a simple function: greetFriend

```
function greetFriend(name) {
  console.log('Hello, ' + name + '!');
}
```

--Call the function:

```
greetFriend('Alice');
// Output: Hello, Alice!
```

Practice Time!
1. Create a function **addNumbers** that takes two numbers and returns their sum.
2. Create a function multiplyNumbers(a, b) that:
    a. Takes two numbers a and b.
    b. Returns their product.

<u>Functions in JavaScript</u>
In JavaScript, functions are reusable blocks of code that perform a specific task. They can take inputs (arguments), process them, and return outputs.

Function Syntax

```
function functionName(parameters) {
  // code to be executed
}
```

Function Types

1. Declared Functions

```
function add(a, b) {
  return a + b;
}
```

2. Function Expressions

```
const add = function(a, b) {
  return a + b;
};
```

3. Arrow Functions (ES6+)

```
const add = (a, b) => {
  return a + b;
};
```

4. Immediately Invoked Function Expressions (IIFE)

```
(function() {
  console.log("This function runs immediately!");
})();
```

Function Components

1. Parameters: Inputs passed to the function.

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}
```

2. Arguments: Values passed to the function when called.

```
greet("John");
```

3. Return Statement: Specifies the output of the function.

```
function add(a, b) {
  return a + b;
}
```

4. Function Body: The code executed within the function.

Function Scope
1. Global Scope: Functions defined outside other functions.
2. Local Scope: Functions defined inside other functions.

Function Closures:
A closure is a function that has access to its own scope and the scope of its outer functions.

```
function outer() {
  let x = 10;
  function inner() {
    console.log(x);
  }
  return inner;
}

const innerFunc = outer();
innerFunc(); // Output: 10
```

Higher-Order Functions
Functions that take other functions as arguments or return functions.

```
function twice(func) {
  return function() {
    func();
    func();
  };
}

function sayHello() {
  console.log("Hello!");
}

const sayHelloTwice = twice(sayHello);
sayHelloTwice(); // Output: Hello! Hello!
```

Callback Functions
Functions passed as arguments to other functions.

```
function setTimeout(callback, delay) {
  // implementation
}

setTimeout(() => {
  console.log("Callback executed!");
}, 2000);
```

Recursion
Functions that call themselves.

```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1);
}
```

Function Binding
Changing the context of a function.

```
function greet() {
  console.log(`Hello, ${this.name}!`);
}

const obj = { name: "John" };
const boundGreet = greet.bind(obj);
boundGreet(); // Output: Hello, John!
```