**Function**

window.alert( );

To run a JavaScript function, we always must include parentheses—*even if there's nothing between them*. When the browser reads the function name and sees the parentheses, it knows to execute the function.

Placing content between the parentheses is called **passing an argument into a function**. For the alert() function, this content will be displayed as the dialog's message whenever the function is called.

Lastly, a semicolon ; closes out the alert() function's code. This semicolon tells the browser that this particular line is completed and any code after it is a new piece of code. Each piece of code, separated by semicolons, is known as an **expression**.

**Comments**

**//** for single line

**/\***

for multi-lines

for multi-lines

for multi-lines

**\*/**

Function

Function Name

Function Keyword

Function parentheses

Function block opening brace

**functiion**

**fight**

**( )**

**{**

Expression

window.alert(“The fight has begun!”);

Value being assigned to variable

**}**

Function block closing brace

Keyword

Assignment operator

Statement terminator

**var**

**playerName**

**=**

**“Tony the Robot”;**

Variable Name

Value being assigned to variable

**Declaring a Function**

When we **declare** a function (i.e., create it and give it a name), we use the keyword function, followed by the name we want to give it. We can name our functions anything we want, but best practice is to name them for their functionality. For example, we named this function fight, but we could instead name it x. We might remember what x does, but will other developers immediately know?

After the functions name, we need to add parentheses. Our fight function doesn't need data passed into it the same way that the alert does, but the parentheses are required regardless.

The curly braces { } wrap the code that belongs to this function. The function will run any code within the braces, and it won't run any code outside them. The code between these curly braces is called a **code block**.

**Calling a Function**

Fight( );

We defined our function but didn't call it. To call the function, add the function name with parentheses and a semicolon at the bottom.

**Storing user input using variables**

Like the alert function we used previously, we can use another window function called a **prompt**. Let's add:

window.prompt(“What is your robot’s name?”);

**String concatenation**

In **string concatenation**, we can write out a string as we typically would, but in order to include variable data, we need to close the string. To do that, put a plus sign + after the closing quotation, then write the variable name. For example:

var playerName = “Tony the Robot”;

console.log(“playerName + “ is ready for battle!”);

console.log(“Your robot, “ + playerName + “, has won!);

**Method-1. Function Declaration:**

//create function

function fight( ) {

window.alert(“Welcome to Robot Gladiators!”);

}

//execute function

Fight( );

**Method-2. Function Expression:**

//create function

var fight = function( ) {

window.alert(“Welcome to Robot Gladiators!”);

}

//execute function

Fight( );

**Declare an Array of Enemy**

var enemyNames = [“Roborto”, “Amy Andriod”, “Robo Trumble”];

console.log(enemyNames);

console.log(enemyNames[0]); //Roborto

console.log(enemyNames[3]); //undefined - the fourth enemy is not there.

console.log(enemyNames.length); //3 with names

**for Loop Syntex**

for([initial expression]; [condition]; [increment expression]) {

statement

}

for(var i = 0; i<3; i++); {

console.log(“apple”);

}

for(var i = 0; i<3; i++); {

console.log(“apple”, i);

}

1. The **initial expression** is the first statement executed, initializing the loop iterator or counter. This expression can also declare variables.
2. The **condition** statement is then evaluated. If this condition evaluates to true, the loop **statement** executes. If the condition's value is false, the for loop terminates. If this condition is omitted, the condition is assumed to be true.
3. Then the **statement** executes. To execute multiple statements, use a block statement { }, as used in if-else statements.
4. The **increment expression** executes, incrementing the iterator, which is the variable i.

Here's an example of an IncrementExpression: i++

This is equivalent to i = i + 1.

1. After the iterator increments, the **condition** is then reevaluated and continues the loop chain.

**For Loops and Arrays**

for(var i = 0; i < enemyNames.length; i++) {

console.log(enemyNames[i]);

console.log(i);

console.log(enemyNames[i] + “ is at “ + i + “ index.”);

};

**Function Argument / Parameter**

console.log(enemyNames); // array argument

alert(“Hello”); // string argument

console.log(enemyNames[i], i); // two arguments, comma separated

variables

function

{

addTwoNumbers(number1, number2)

number1 + number2;

return

to create result

};

//10

addTwoNumbers(4, 6);

value

var myCar = {

make: “Toyota",

model: “Prius”,

mileage: 20000,

driveToWork: function(){

this.mileage += 25;

}

};

myCar.driveToWork();

console.log(myCar);

//console.log(myCar.make);

//myCar.color = “grey”;

//console.log(myCar);

fight(enemyRobot);

var fight = function(enemyName) {

…// fight function statements

}

Parameters often get confused with arguments because their syntax is similar. The main distinction between them is their purpose in the function. In the following function expression, a parameter serves as a variable placeholder that indicates how the variable will be used in the function. Because the parameter is only used for the scope of the function, its name isn't particularly important—but it typically relates to the purpose of the variable, as you can see in the following code:

var wash = function(soapType) {

console.log(“I wash with “ + soapType);

};

Wash(“Spring.”); // => I wash with Spring.

As part of the function call, the argument passes information into the function—whether that be a variable or a value—with any data type or structure.

for(var i = 0; i < enemyNames.length; i++) {

fight(enemyNames[i]);

}

Notice that the fight() function call was replaced with a for loop that calls the fight() function multiple times using the element in the enemyNames[i] array as the argument.

**while Loop Syntex**

We can also introduce another type of control flow statement that loops or repeatedly executes a statement while a condition remains true. This is called the while loop. Like the for loop, the while loop repeatedly executes a code block only if a condition remains true.

while([condition]) {

statement

}

var fight = function(enemyName) {

// repeat and execute as long as the enemy-robot is alive

while(enemyHealth > 0) {

// place fight function code block here . . .

}

**Use the Debugger**

for (var i = 0; i < enemyNames.length; i++) {

debugger;

// call fight function with enemy-robot

fight(enemyNames[i]);

}

for (var i = 0; i < enemyNames.length; i++) {

var pickedEnemyName = enemyNames[i];

enemyHealth = 50;

fight(pickedEnemyName);

}

if (playerHealth <= 0) {

window.alert(playerName + " has died!");

break;

}

while (playerHealth > 0 && enemyHealth > 0)

With the logical AND operator (&&), we can have the while loop set two conditions that must both resolve to true to execute the fighting rounds in the while loop. The AND operator differs from the OR operator (||), which must have either condition evaluate to true. The AND operator must satisfy both conditions to execute the block.

// if yes (true), leave fight

if (confirmSkip) {

window.alert(playerName + " has decided to skip this fight. Goodbye!");

// subtract money from playerMoney for skipping

playerMoney = playerMoney - 10;

console.log("playerMoney", playerMoney);

}

// if no (false), ask question again by running fight() again

Else fight();

// if player picks "skip" confirm and then stop the loop

if (promptFight === "skip" || promptFight === "SKIP") {

// confirm player wants to skip

var confirmSkip = window.confirm("Are you sure you'd like to quit?");

// if yes (true), leave fight

if (confirmSkip) {

window.alert(playerName + " has decided to skip this fight. Goodbye!");

// subtract money from playerMoney for skipping

playerMoney = playerMoney - 10;

console.log("playerMoney", playerMoney)

break;

}

}

var startGame = function() {

for (var i = 0; i < enemyNames.length; i++) {

...

}

// play again

startGame();

};

Variable Scope

Note that the startGame() function is allowed to read and update these three variables because of the **scope** they were declared in. Variables like playerHealth that are declared outside of any functions are considered **global**, meaning that any function can access them.

var pickedEnemyName = enemyNames[i];

The pickedEnemyName variable only exists within the scope of the startGame() function, so other functions like fight() can't access it. This is called **local scope**.

var a = "a";

var logStuff = function() {

var b = "b";

console.log(a);

console.log(b);

};

console.log(a);

console.log(b); // error

The variable a was declared outside any function, making it global in scope. Therefore, the logStuff() function can access it. The variable b was declared inside a function, so only that function has access to it. If we were to try to access b outside of logStuff(), we'd get an error.

Scope presents a good case for writing unique, meaningful variable names. If a global variable and a local function variable have the same name, the local variable will take precedence. To avoid such confusion, it's best not to reuse variable names.

// ask player if they'd like to play again

var playAgainConfirm = window.confirm("Would you like to play again?");

if (playAgainConfirm) {

// restart the game

startGame();

}

else {

window.alert("Thank you for playing Robot Gladiators! Come back soon!");

}

**Math Object**

// prints 100

console.log(Math.max(10, 20, 100));

// prints 0

console.log(Math.max(0, -50));

enemyHealth = Math.floor(Math.random() \* 60);

By multiplying Math.random() by 60, we've now specified a random range from 0 to 59.xx (remember, Math.random() will never be 1, so we would never get an even 60). We don't want decimal numbers cluttering up our game, though, so we can use Math.floor() to round down to the nearest whole number. This means that at the start of each round, enemyHealth would be a random whole number from 0 to 59.