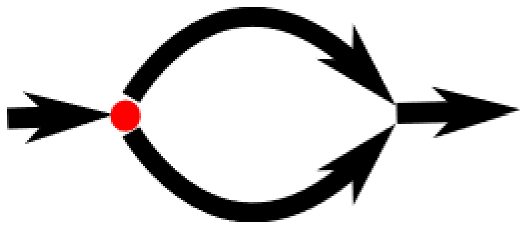
Control flow Structures

Control flow

- When your program contains more than one statement, the statements are executed as if they are a story, from top to bottom.

Control flow

Conditional Execution (creating branches)



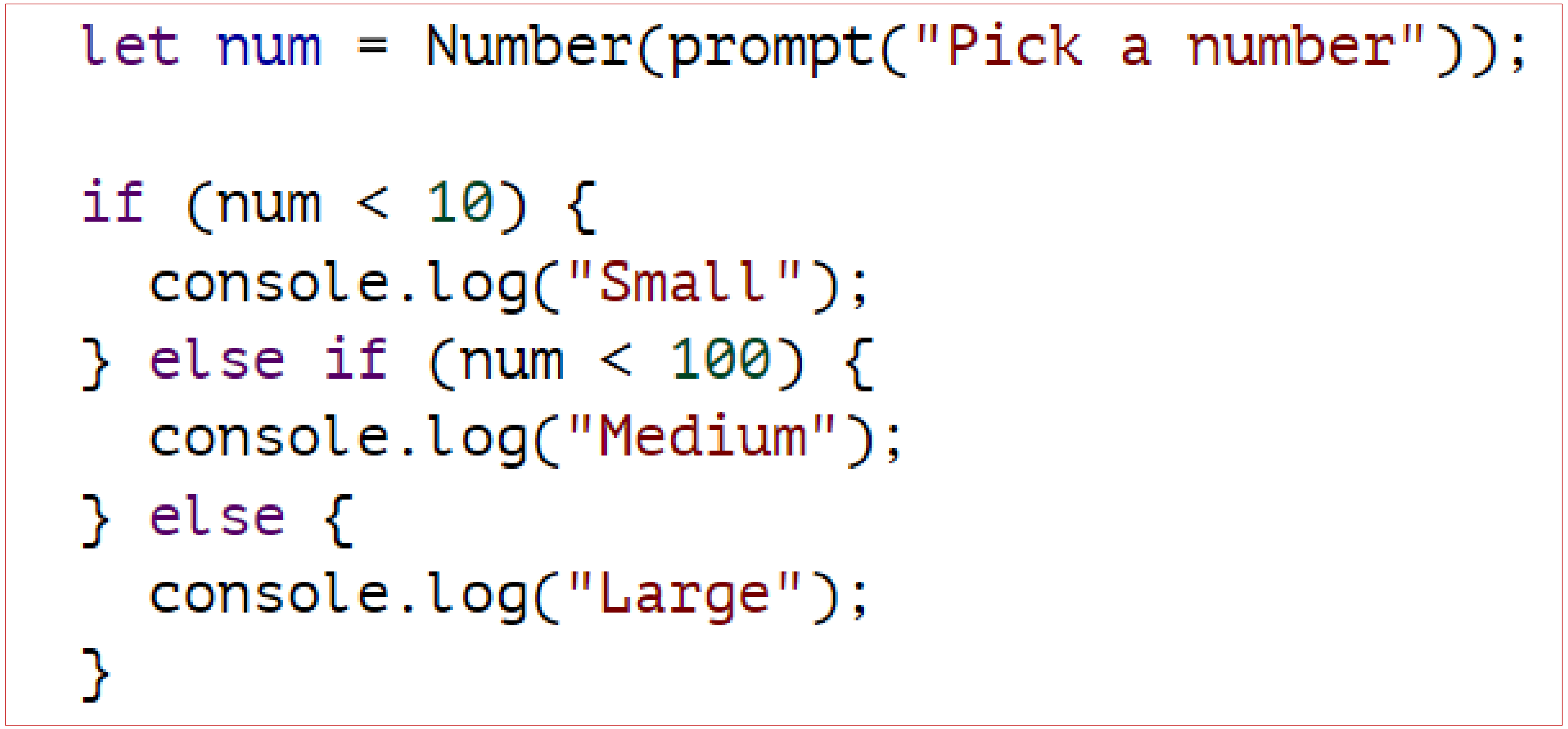
```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
}
```

```
if (1 + 1 == 2) console.log("It's true");
// → It's true
```

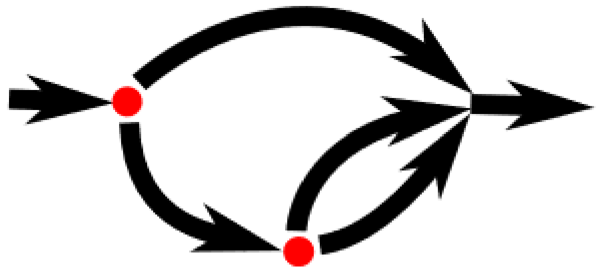
```
let theNumber = Number(prompt("Pick a number"));
if (!Number.isNaN(theNumber)) {
  console.log("Your number is the square root of " +
              theNumber * theNumber);
} else {
  console.log("Hey. Why didn't you give me a number?");
}
```

Control flow

Conditional Execution (creating branches)

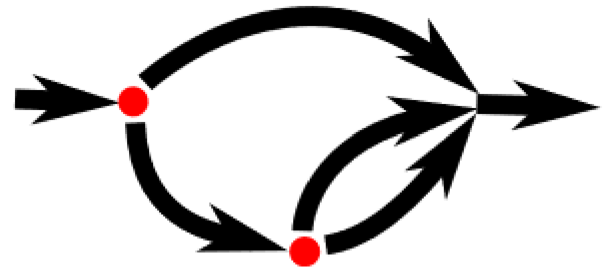


```
let num = Number(prompt("Pick a number"));

if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```



Control flow

Conditional Execution (creating branches)

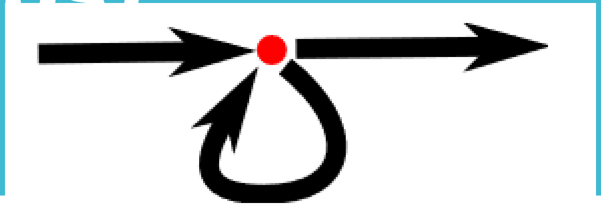


```
if (x == "value1") action1();
else if (x == "value2") action2();
else if (x == "value3") action3();
else defaultAction();
```

```
switch (prompt("What is the weather like?")) {
  case "rainy":
    console.log("Remember to bring an umbrella.");
    break;
  case "sunny":
    console.log("Dress lightly.");
  case "cloudy":
    console.log("Go outside.");
    break;
  default:
    console.log("Unknown weather type!");
    break;
}
```

Control flow

Conditional Execution (creating repetition/loops)



```
let number = 0;
while (number <= 12) {
  console.log(number);
  number = number + 2;
}
// → 0
// → 2
//   … etcetera
```

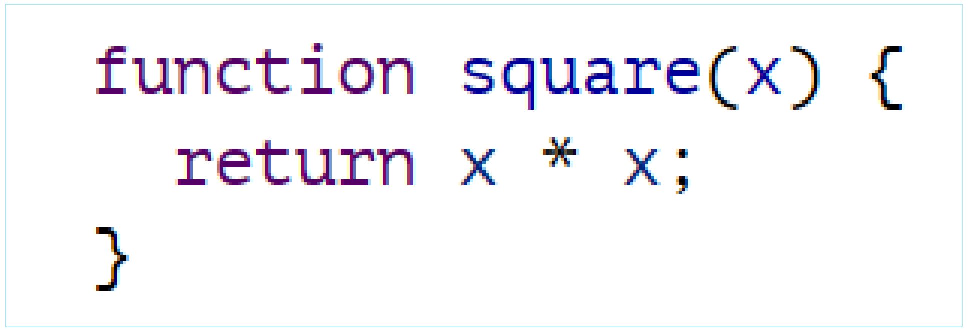
```
let yourName;
do {
  yourName = prompt("Who are you?");
} while (!yourName);
console.log(yourName);
```

```
for (let number = 0; number <= 12; number = number + 2) {
  console.log(number);
}
// → 0
// → 2
//   … etcetera
```

JS Functions

Function Declaration notation

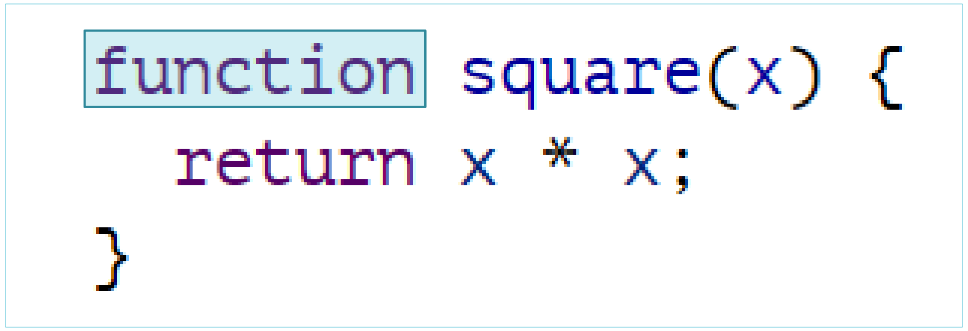
Function Declaration notation



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

- Using the ‘**function**’ keyword at the beginning of the line.
- The name of he function follows the identifiers rules.
- The list of comma separated parameters surrounded by round parentheses.
- The body of the function must be surrounded by curly brackets even if it was one statement.
- If we don’t use the ‘**return**’ keyword the function will return **undefined.**

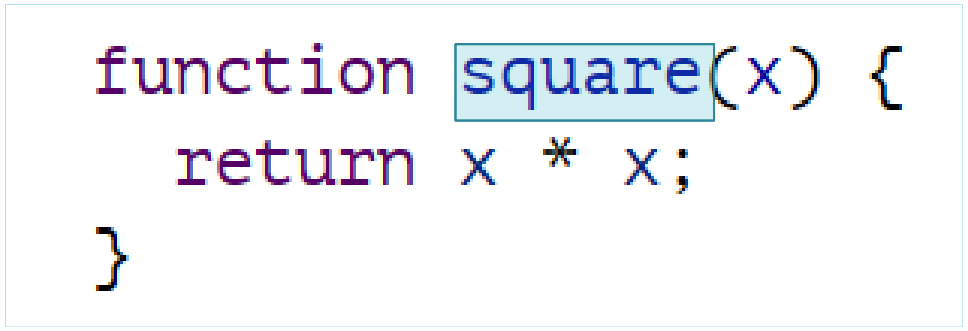
Function Declaration notation



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

- Using the ‘**function**’ keyword at the beginning of the line.
- The name of he function follows the identifiers rules.
- The list of comma separated parameters surrounded by round parentheses.
- The body of the function must be surrounded by curly brackets even if it was one statement.
- If we don’t use the ‘**return**’ keyword the function will return **undefined.**

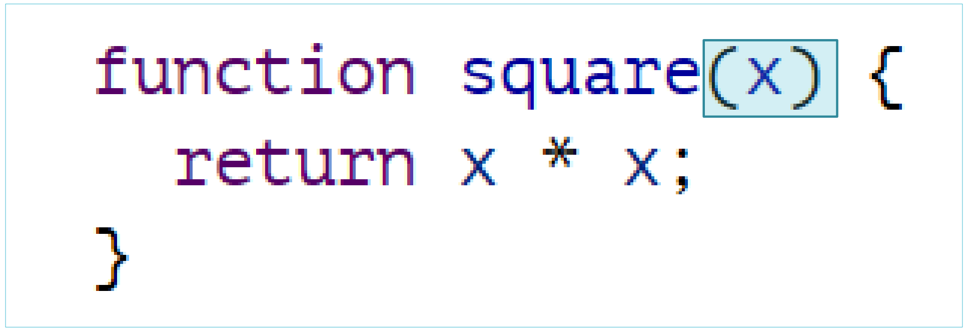
Function Declaration notation



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

- Using the ‘**function**’ keyword at the beginning of the line.
- The name of he function follows the identifiers rules.
- The list of comma separated parameters surrounded by round parentheses.
- The body of the function must be surrounded by curly brackets even if it was one statement.
- If we don’t use the ‘**return**’ keyword the function will return **undefined.**

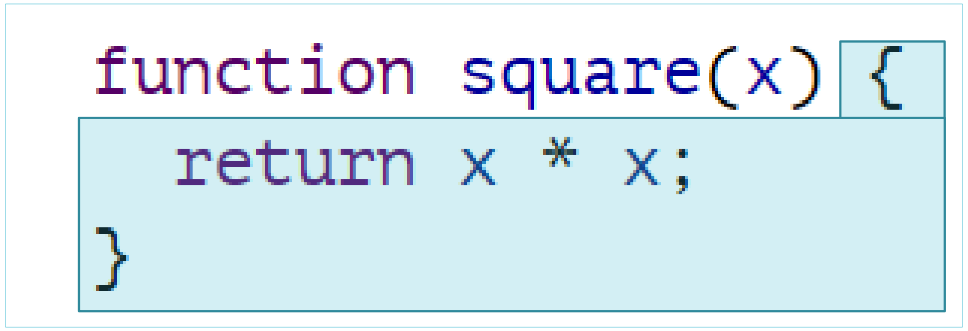
Function Declaration notation



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

- Using the ‘**function**’ keyword at the beginning of the line.
- The name of he function follows the identifiers rules.
- The list of comma separated parameters surrounded by round parentheses.
- The body of the function must be surrounded by curly brackets even if it was one statement.
- If we don’t use the ‘**return**’ keyword the function will return **undefined.**

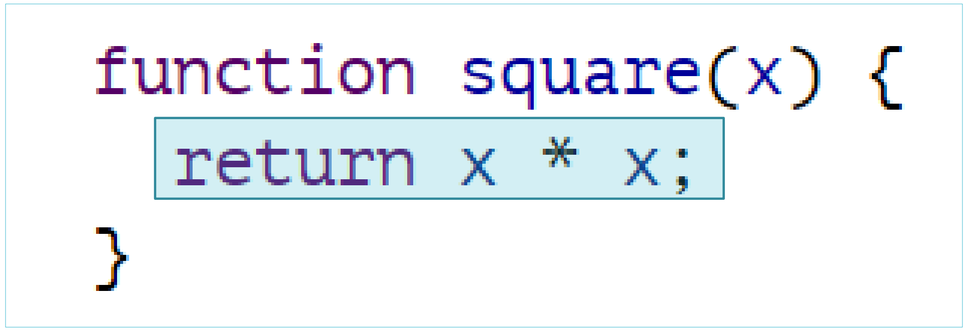
Function Declaration notation



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

- Using the ‘**function**’ keyword at the beginning of the line.
- The name of he function follows the identifiers rules.
- The list of comma separated parameters surrounded by round parentheses.
- The body of the function must be surrounded by curly brackets even if it was one statement.
- If we don’t use the ‘**return**’ keyword the function will return **undefined.**

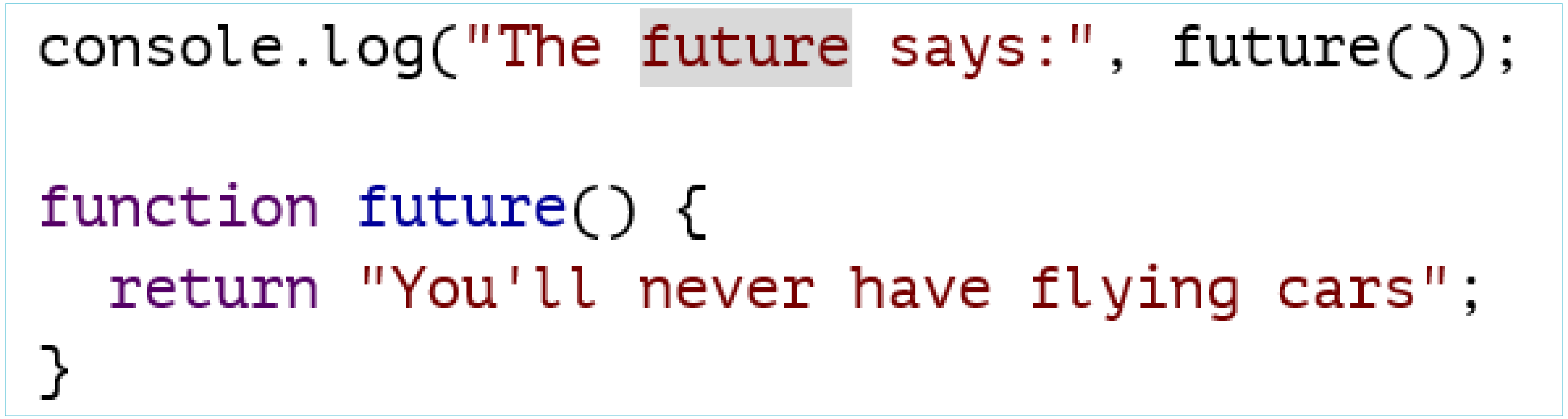
Function Declaration notation



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

- Using the ‘**function**’ keyword at the beginning of the line.
- The name of he function follows the identifiers rules.
- The list of comma separated parameters surrounded by round parentheses.
- The body of the function must be surrounded by curly brackets even if it was one statement.
- If we don’t use the ‘**return**’ keyword the function will return **undefined.**

Function Declaration notation



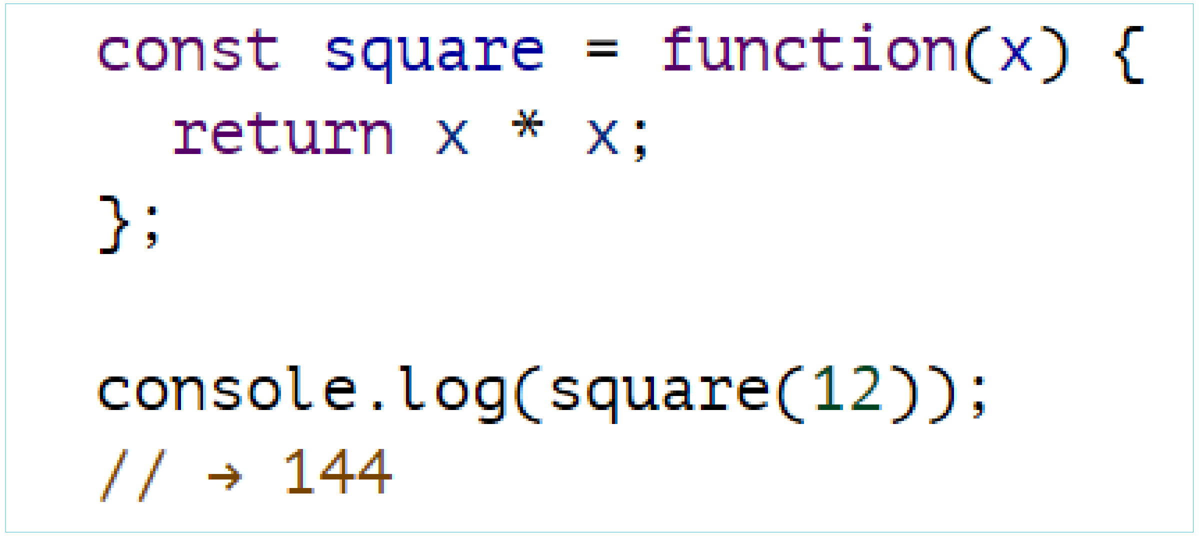
```
console.log("The future says:", future());

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

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

Function Expressions & bindings

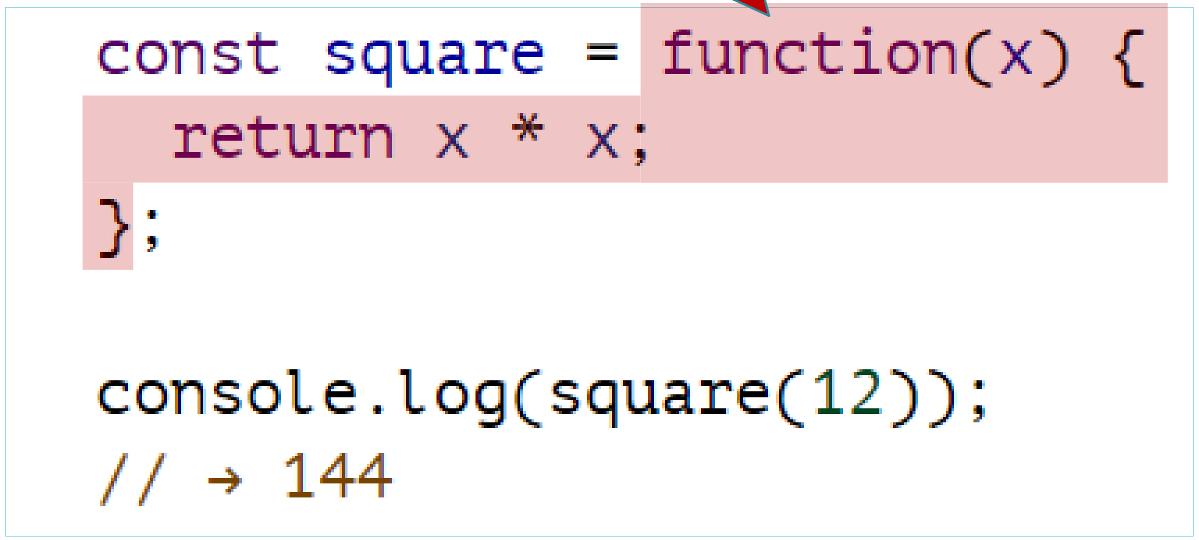
Function Expressions & Bindings



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

console.log(square(12));
// → 144
```

Function Expressions & Bindings

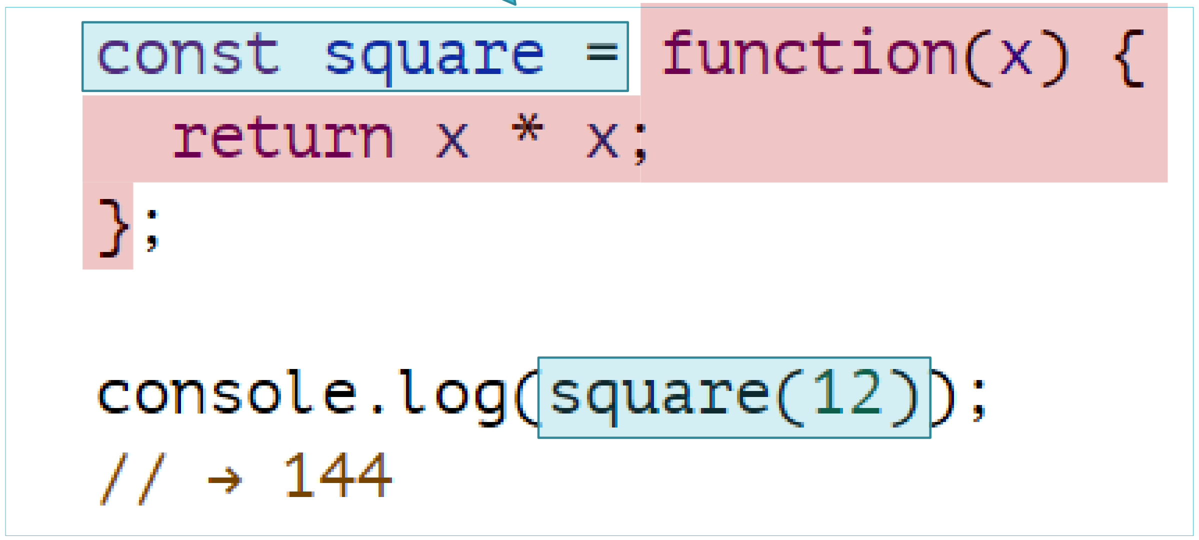
This part by itself is called anonymous function expression which can be treated like any value



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

console.log(square(12));
// → 144
```

Function Expressions & Bindings

When assigned to a binding, the name of the binding can be used as the function name

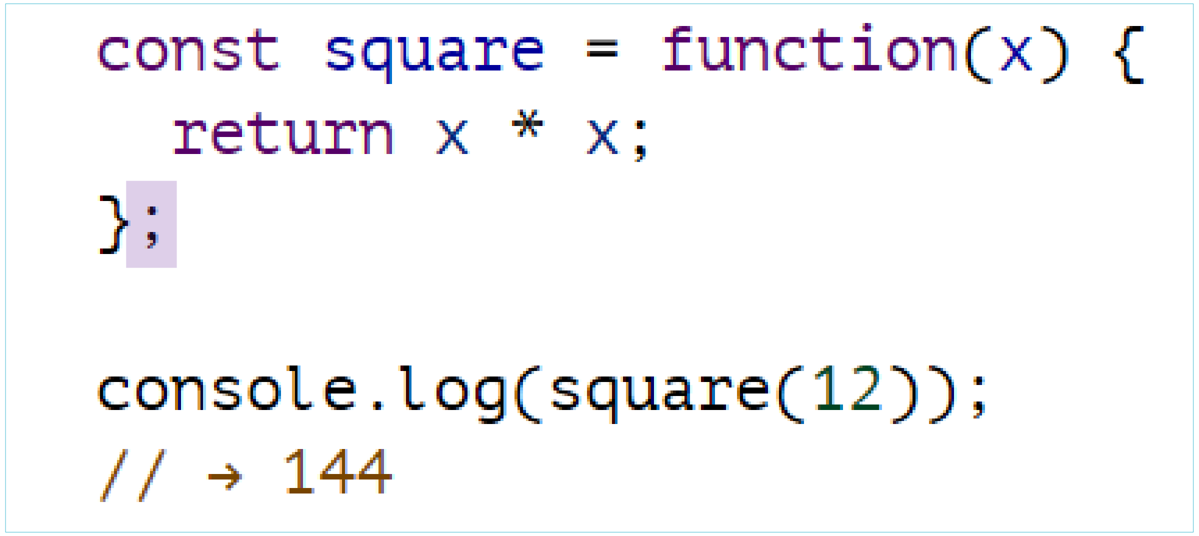


```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

Function Expressions & Bindings

This way of defining function in JS differs from the formal declaration notation in that:

- A semicolon is needed to mark the end of the statement
- The line order is important

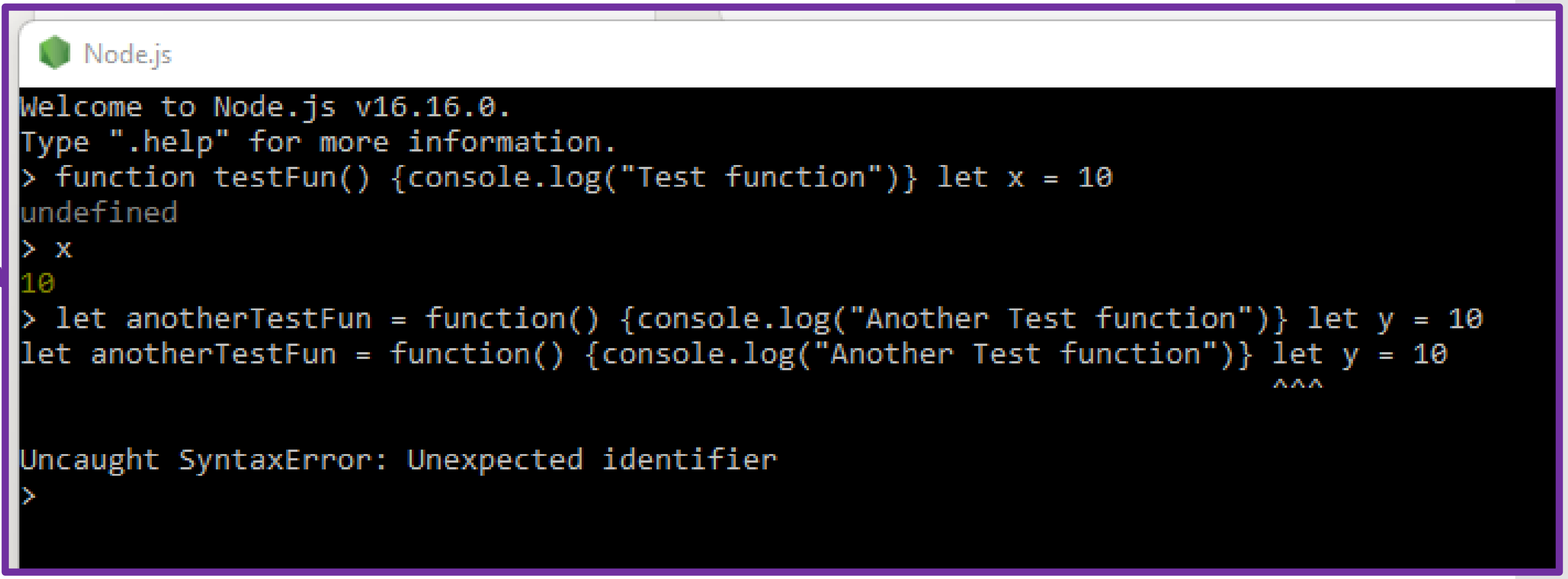


```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

Function Expressions & Bindings

This way of defining function in JS differs from the formal declaration notation in that:

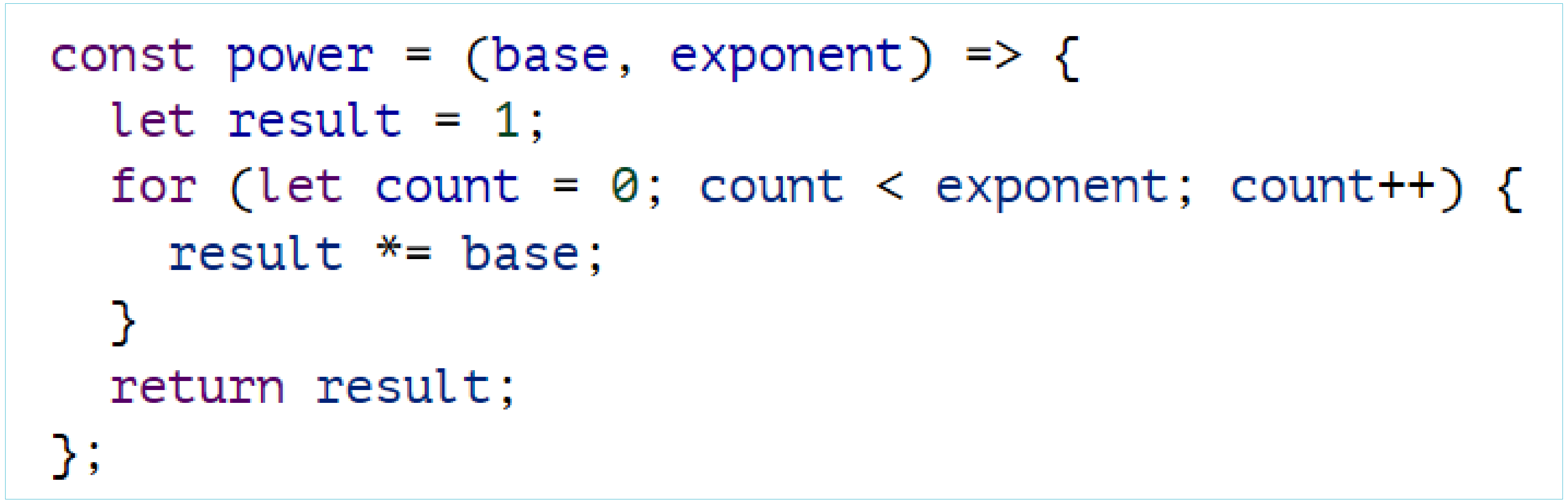
- A semicolon is needed to mark the end of the statement
- The line order is important



```
Welcome to Node.js v16.16.0.
Type ".help" for more information.
> function testFun() {console.log("Test function")} let x = 10
undefined
> x
10
> let anotherTestFun = function() {console.log("Another Test function")} let y = 10
let anotherTestFun = function() {console.log("Another Test function")} let y = 10
                                                                       ^^^

Uncaught SyntaxError: Unexpected identifier
>
```

Arrow Functions

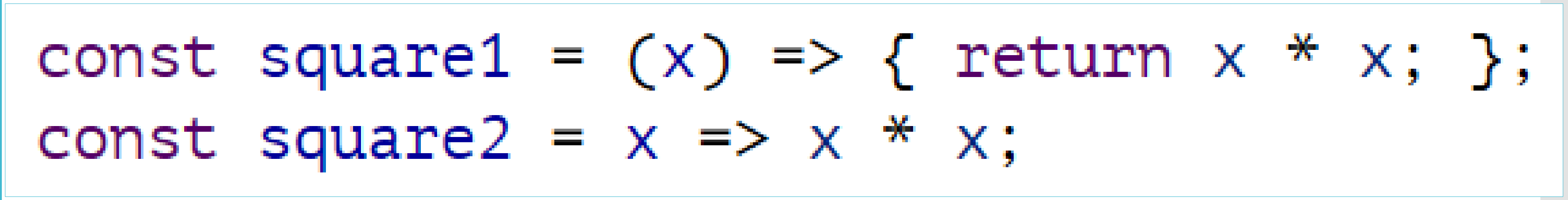
Arrow functions



```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

- Instead of the **function** keyword, we use an arrow (=>) made up of an equal sign and a greater-than character
- The arrow comes after the list of parameters and is followed by the function’s body.

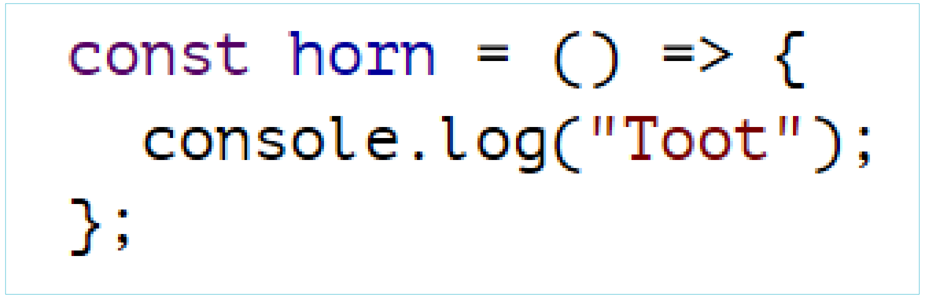
Arrow functions



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

- When there is only one parameter name, you can omit the parentheses around the parameter list.
- If the body is a single expression in a return statement, we can omit the curly brackets and the return keyword.

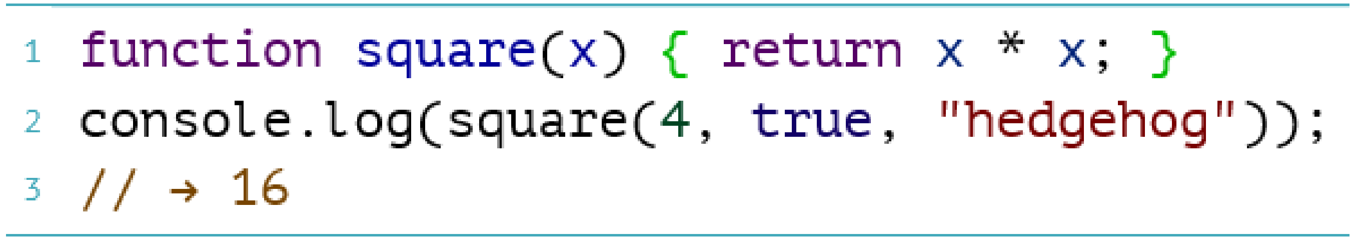
Arrow functions



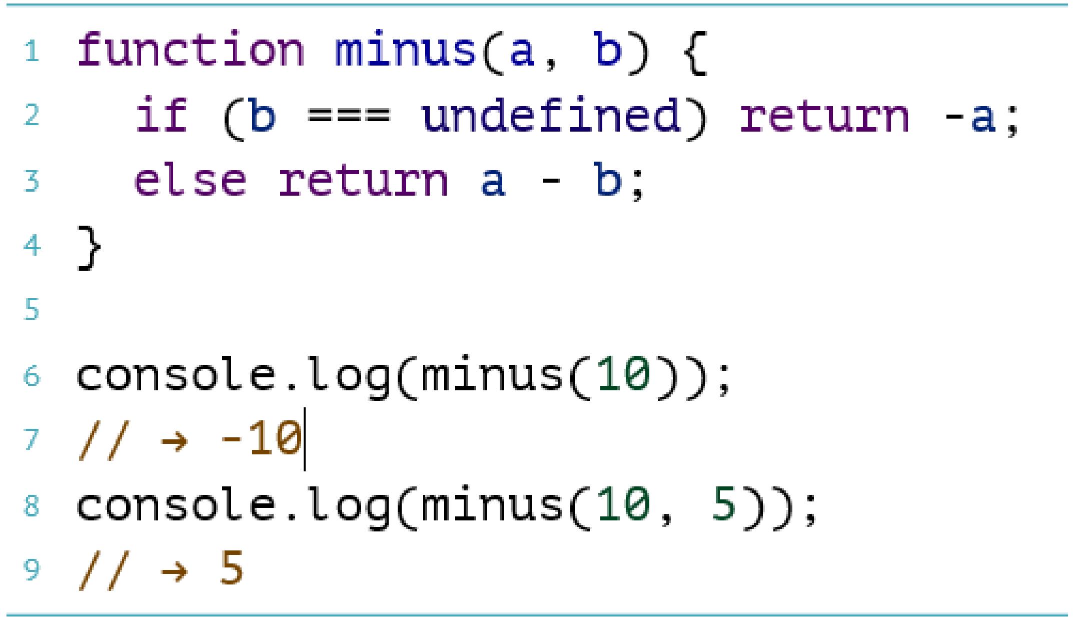
```
const horn = () => {
  console.log("Toot");
};
```

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

Optional Arguments



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

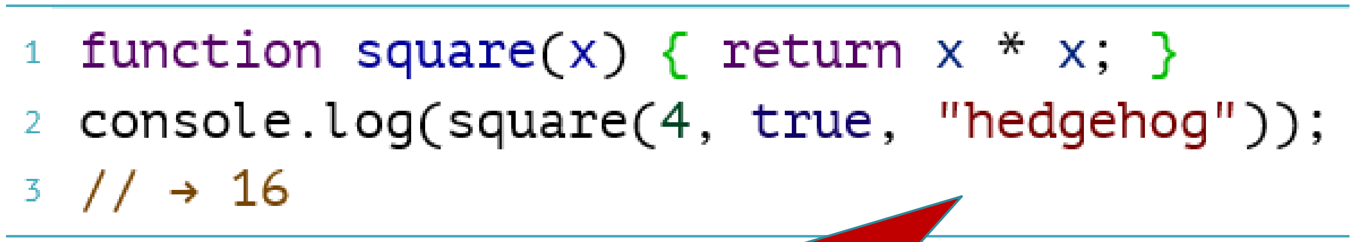


```javascript
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

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

Optional Arguments

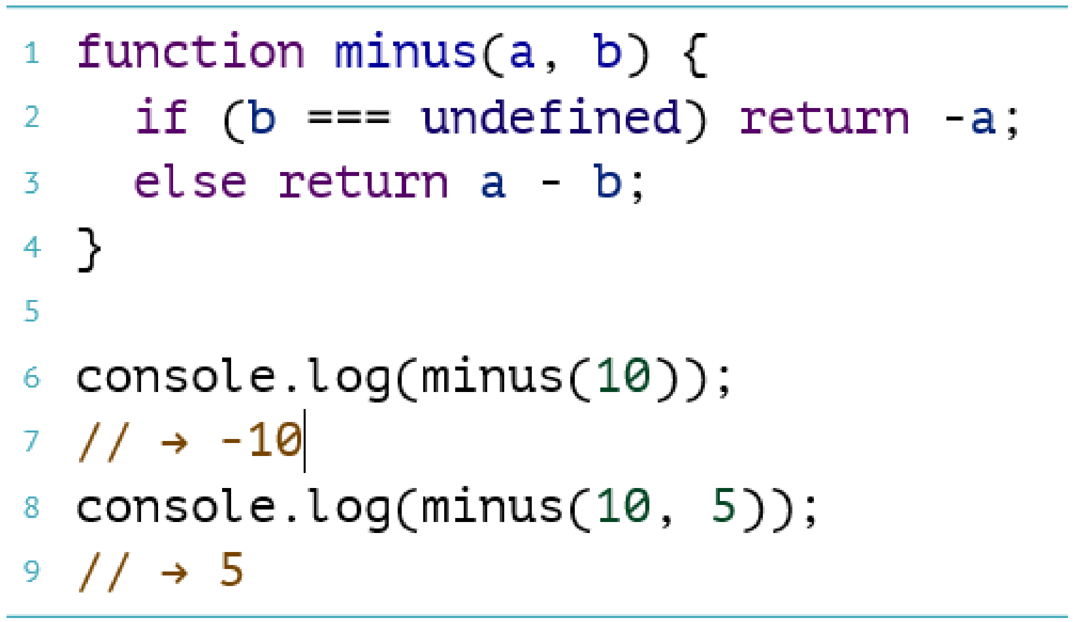
When we define a function we can specify a list of parameters

In JavaScript, the caller is not restricted by the number of parameters in this list



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

Extra parameters will be ignored



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

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

Optional Argumen ts

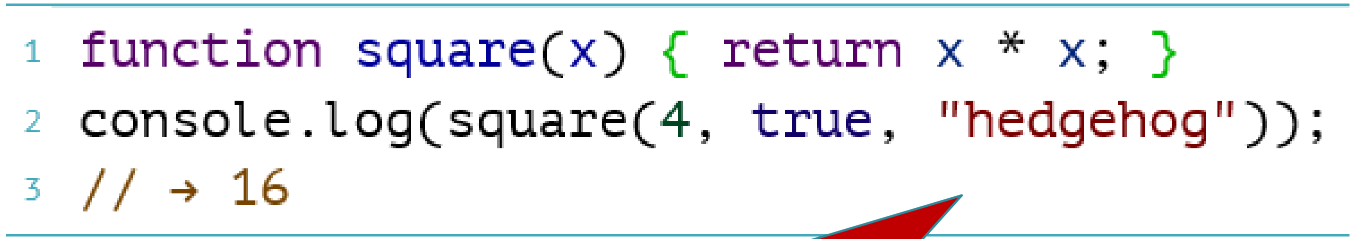
When we define a function we can specify a list of parameters

In JavaScript, the caller is not restricted by the number of parameters in this list

Optional Arguments

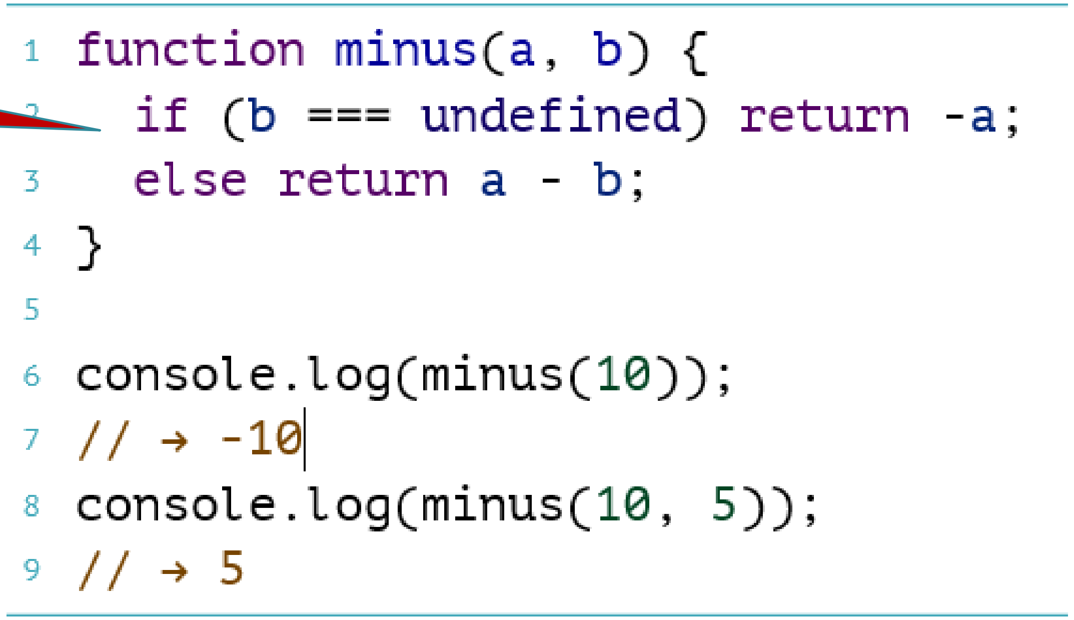
When we define a function we can specify a list of parameters

In JavaScript, the caller is not restricted by the number of parameters in this list



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

Extra parameters will be ignored

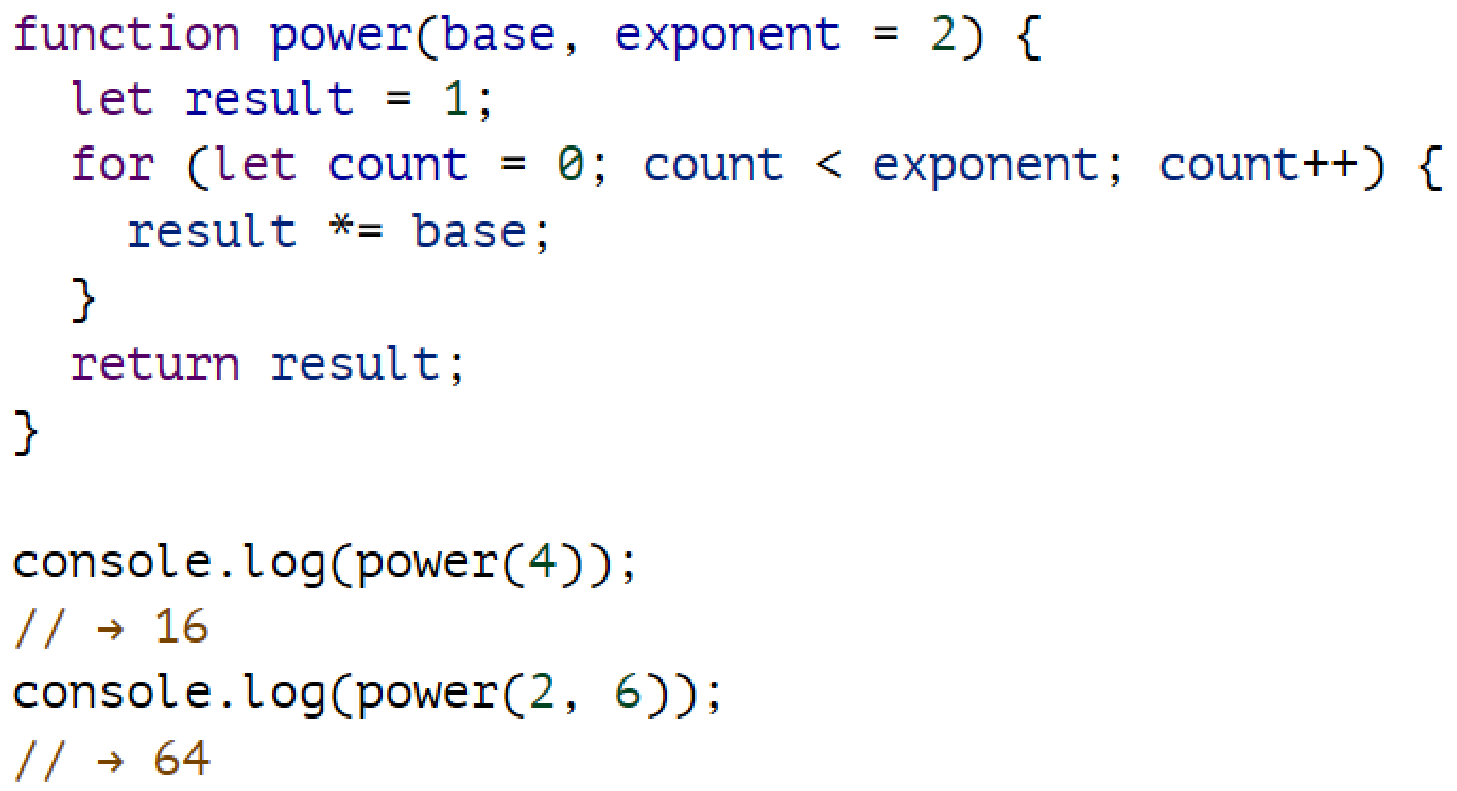


```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

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

Missing parameters will be set to undefined

Optional Arguments

If we give the parameter a default value, the parameter will be assigned that default value when omitted in the function call



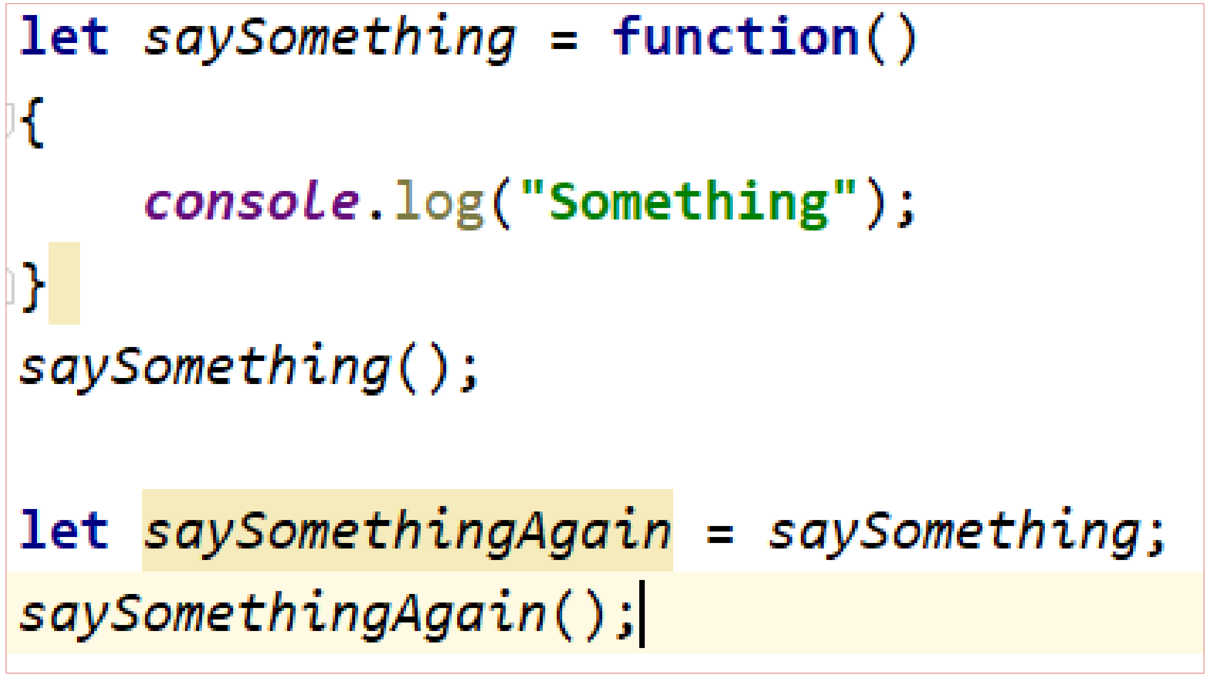
```
function power(base, exponent = 2) {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
}

console.log(power(4));
// → 16
console.log(power(2, 6));
// → 64
```

Functions as values

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions

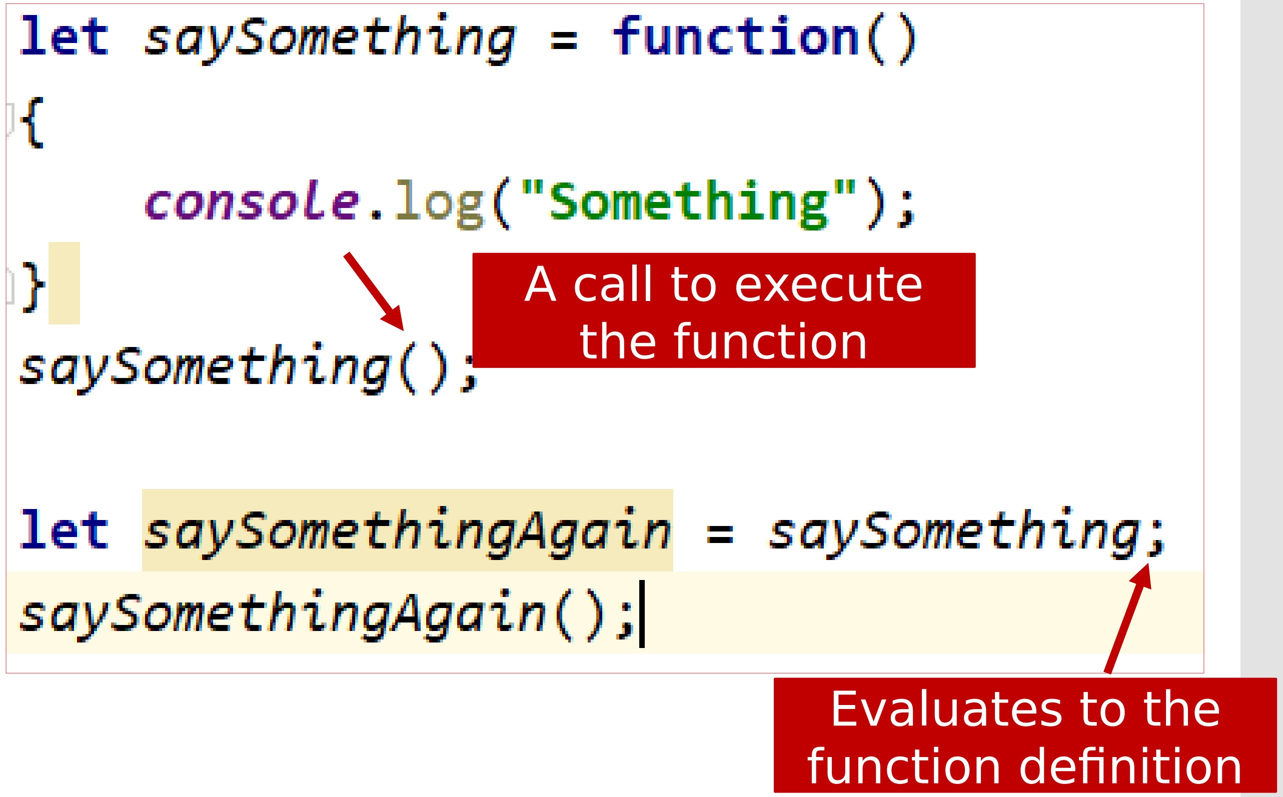


```
let saySomething = function()
{
    console.log("Something");
}
saySomething();

let saySomethingAgain = saySomething;
saySomethingAgain();
```

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions



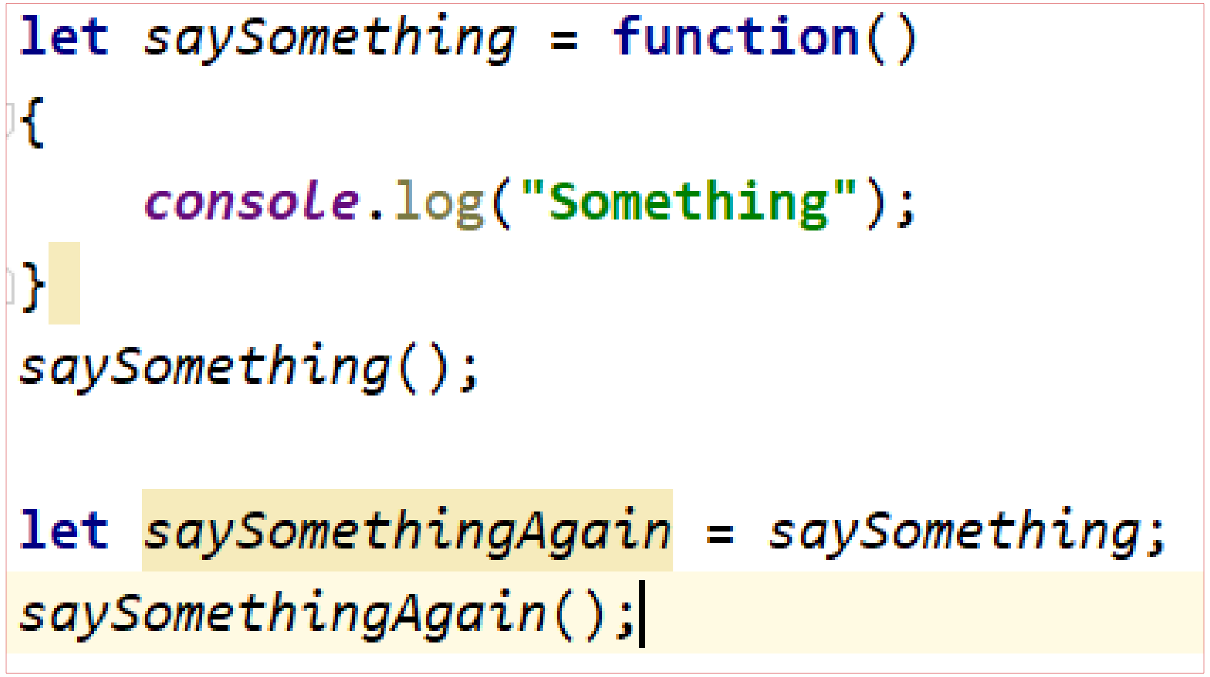
```
let saySomething = function()
{
    console.log("Something");
}
saySomething();

let saySomethingAgain = saySomething;
saySomethingAgain();
```

A call to execute the function
Evaluates to the function definition

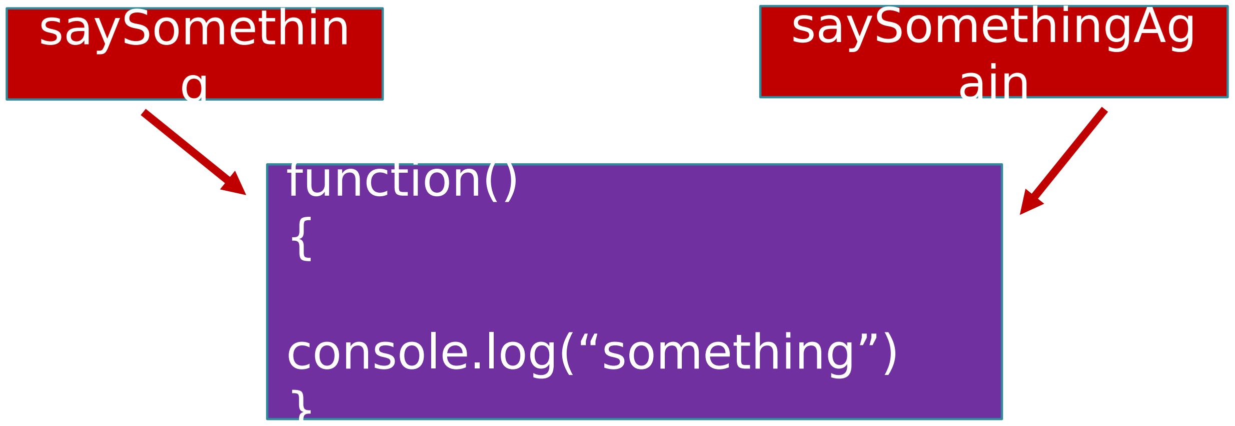
Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions



```
let saySomething = function()
{
    console.log("Something");
}
saySomething();

let saySomethingAgain = saySomething;
saySomethingAgain();
```

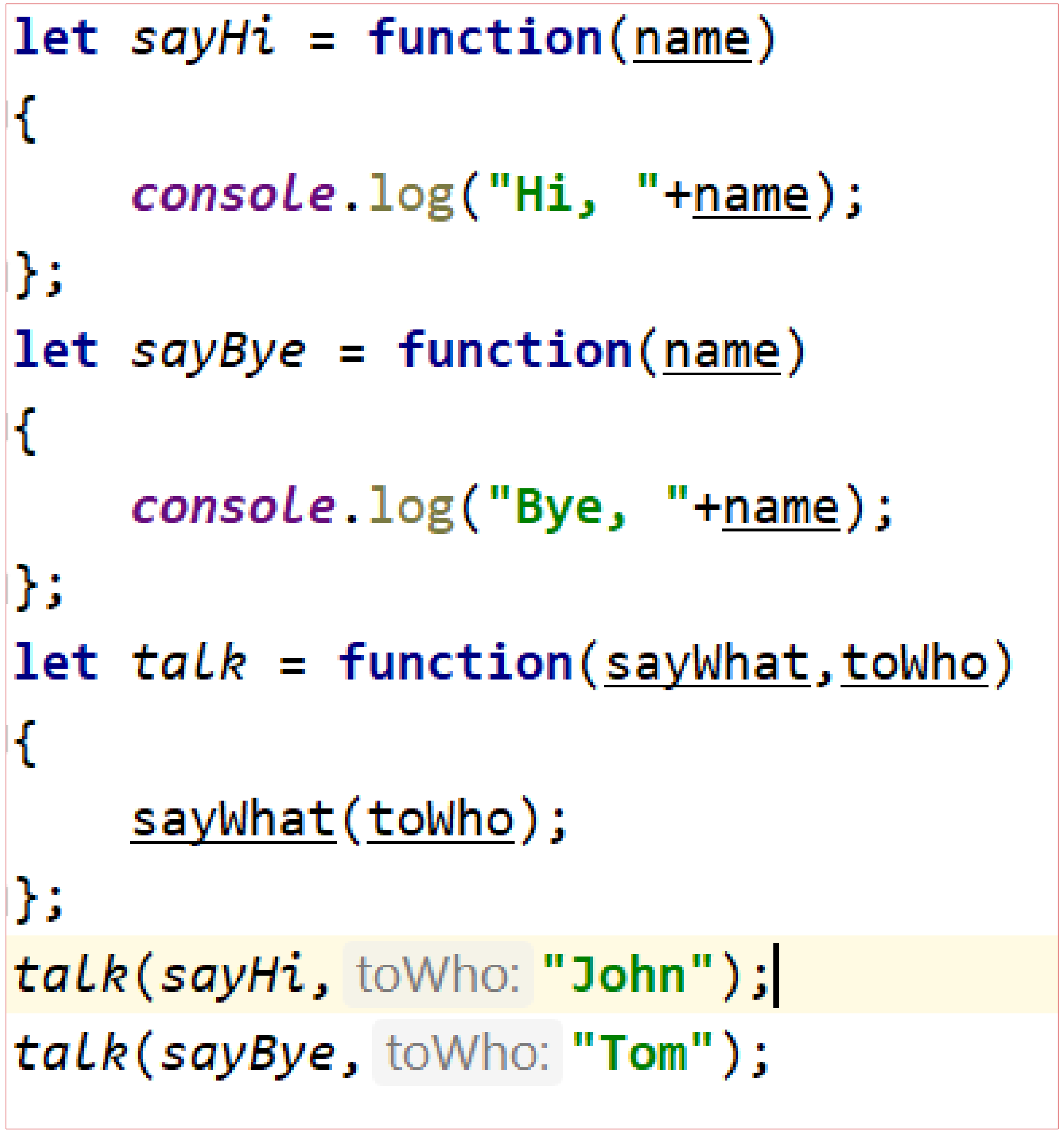


```
function()
{

console.log(“something”)
}
```

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions



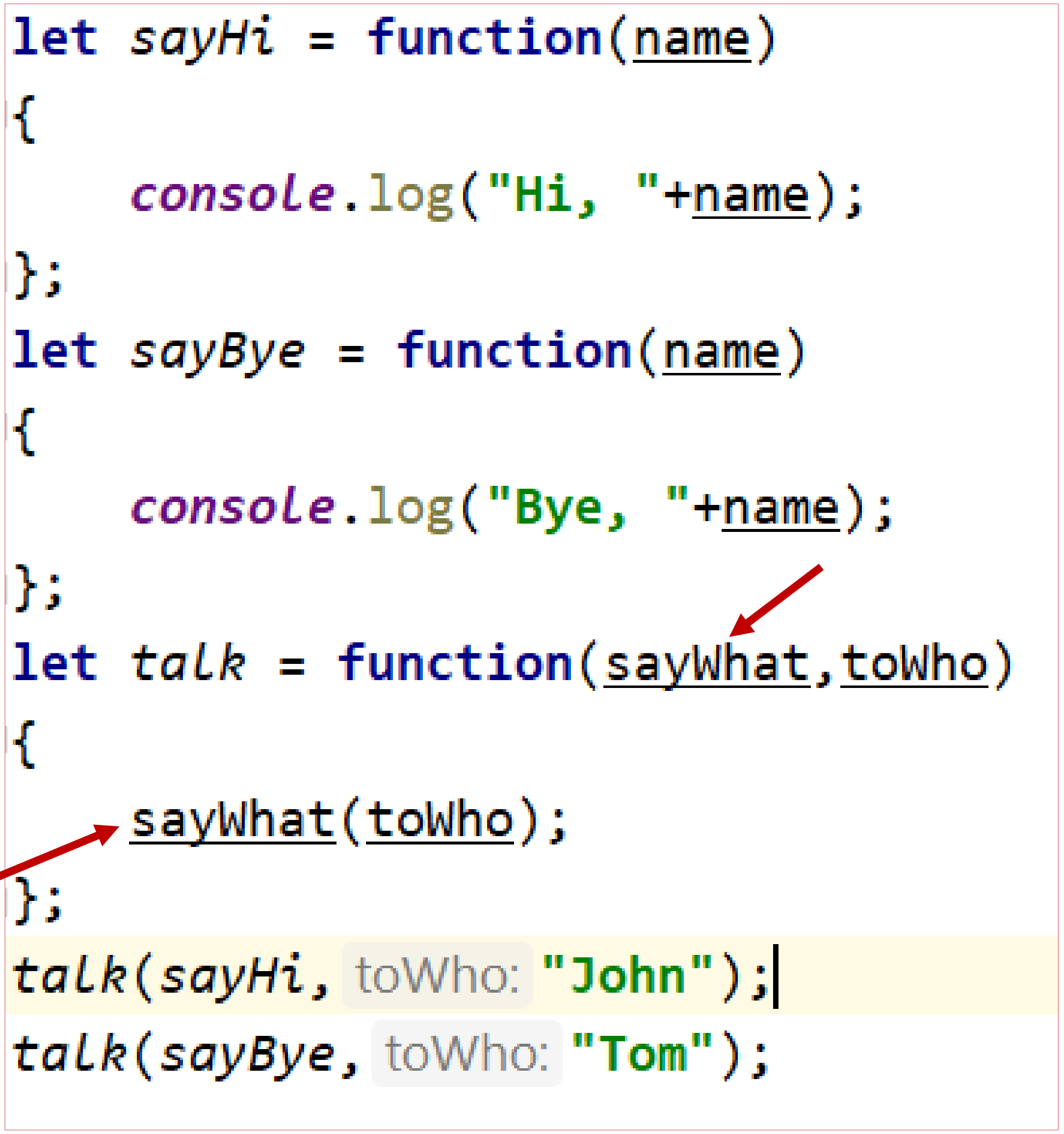
```
let sayHi = function(name)
{
    console.log("Hi, "+name);
};
let sayBye = function(name)
{
    console.log("Bye, "+name);
};
let talk = function(sayWhat,toWho)
{
    sayWhat(toWho);
};
talk(sayHi, toWho: "John");
talk(sayBye, toWho: "Tom");
```

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
  The function is passed in order to be called back
- returned from functions

Callback
function

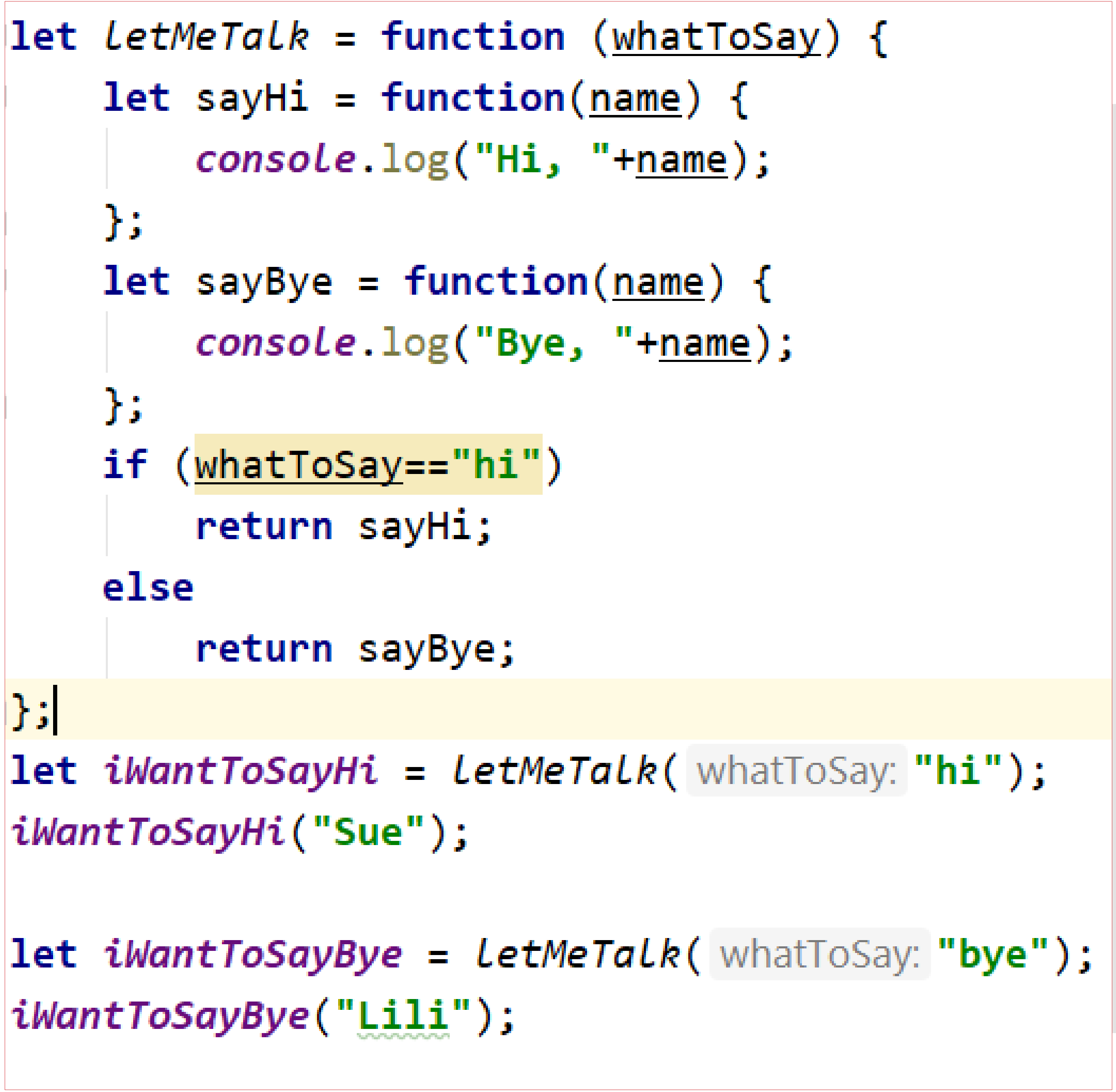


```
let sayHi = function(name)
{
    console.log("Hi, "+name);
};
let sayBye = function(name)
{
    console.log("Bye, "+name);
};
let talk = function(sayWhat,toWho)
{
    sayWhat(toWho);
};
talk(sayHi, toWho: "John");
talk(sayBye, toWho: "Tom");
```

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions



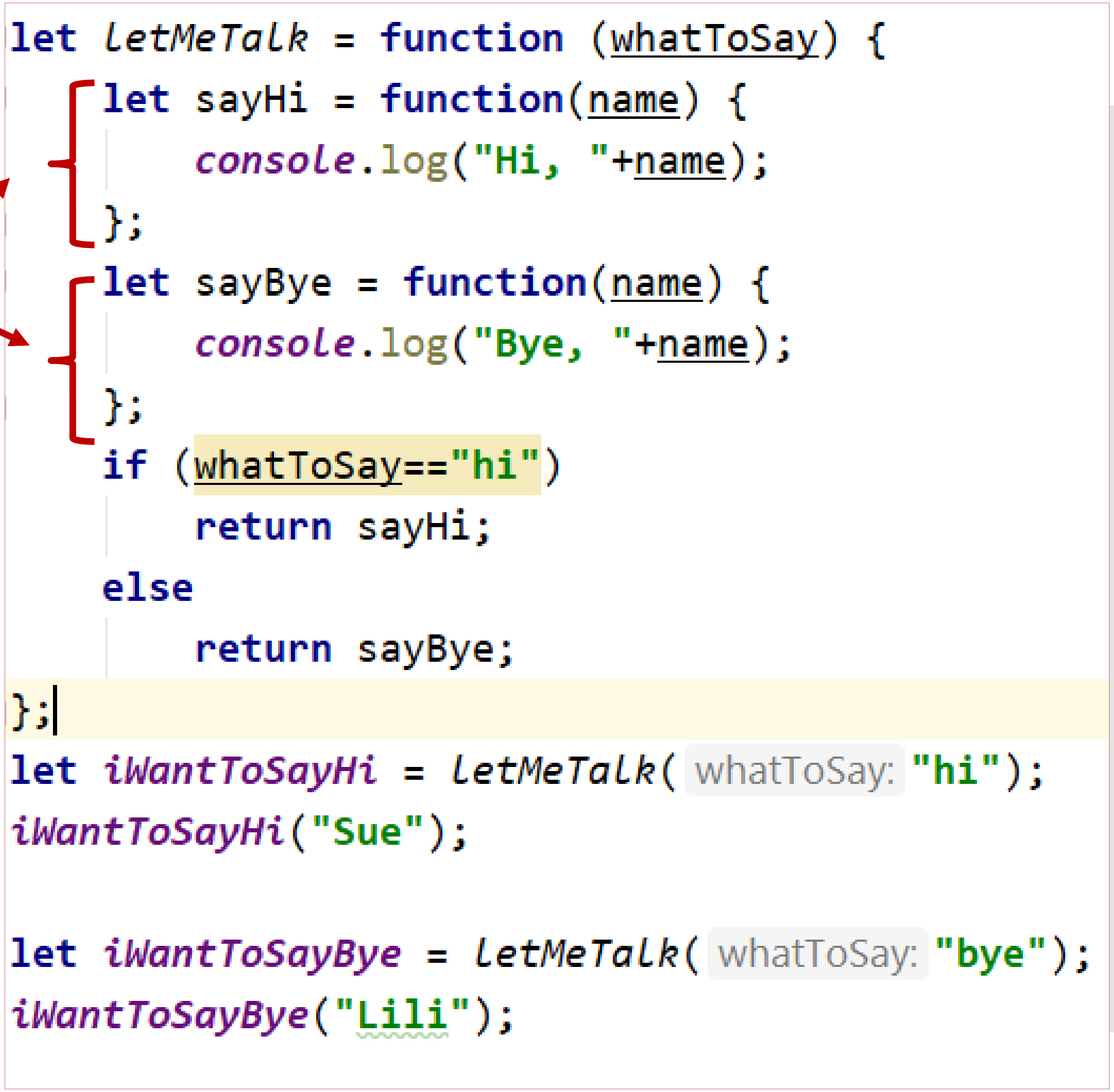
```
let letMeTalk = function (whatToSay) {
    let sayHi = function(name) {
        console.log("Hi, "+name);
    };
    let sayBye = function(name) {
        console.log("Bye, "+name);
    };
    if (whatToSay=="hi")
        return sayHi;
    else
        return sayBye;
};
let iWantToSayHi = letMeTalk( whatToSay: "hi");
iWantToSayHi("Sue");

let iWantToSayBye = letMeTalk( whatToSay: "bye");
iWantToSayBye("Lili");
```

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions

Inner function

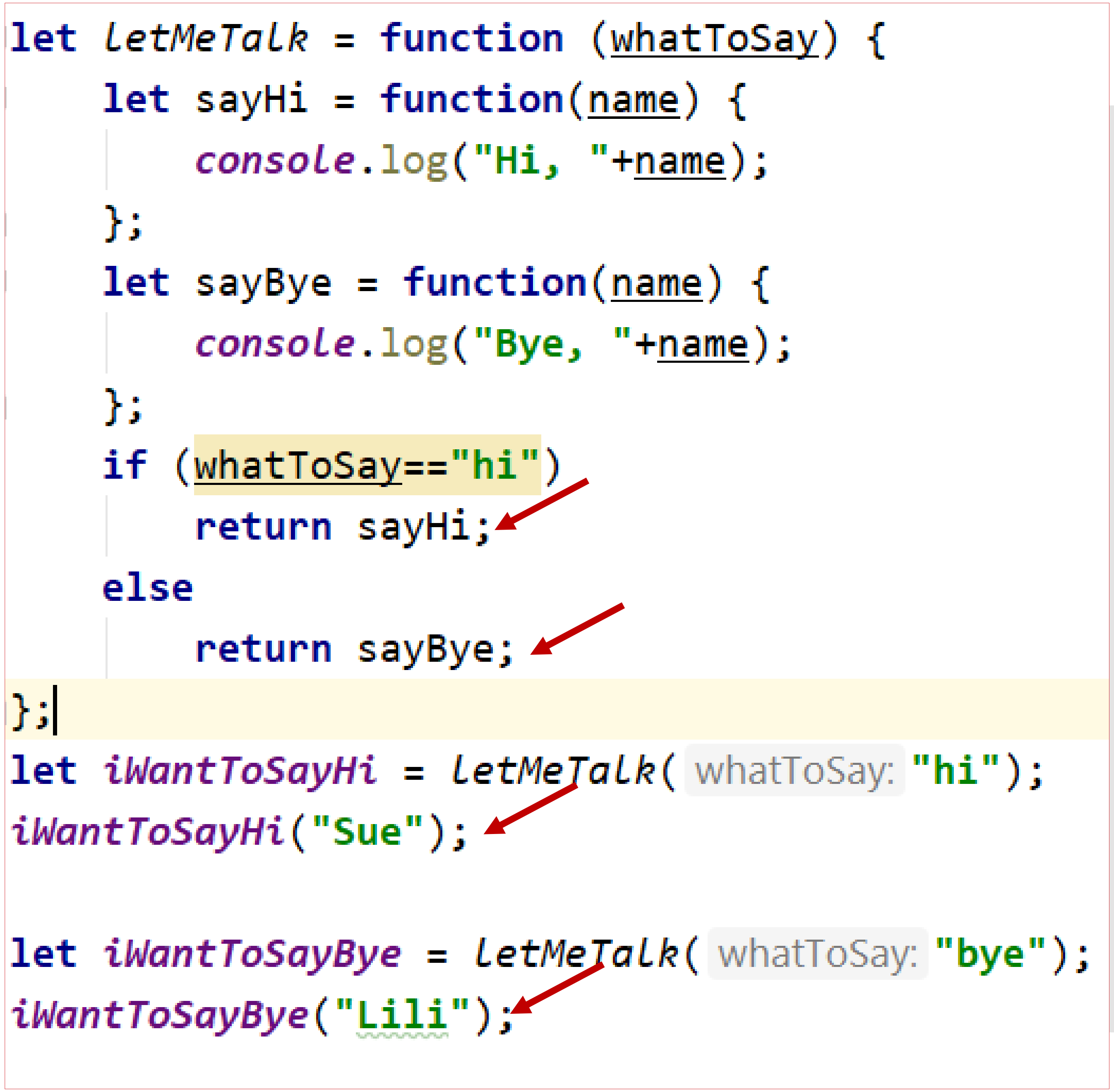


```
let letMeTalk = function (whatToSay) {
    let sayHi = function(name) {
        console.log("Hi, "+name);
    };
    let sayBye = function(name) {
        console.log("Bye, "+name);
    };
    if (whatToSay=="hi")
        return sayHi;
    else
        return sayBye;
};
let iWantToSayHi = letMeTalk( whatToSay: "hi");
iWantToSayHi("Sue");

let iWantToSayBye = letMeTalk( whatToSay: "bye");
iWantToSayBye("Lili");
```

Functions as values

Can be:

- re-assigned to different bindings
- passed as parameters
- returned from functions



```
let letMeTalk = function (whatToSay) {
    let sayHi = function(name) {
        console.log("Hi, "+name);
    };
    let sayBye = function(name) {
        console.log("Bye, "+name);
    };
    if (whatToSay=="hi")
        return sayHi;
    else
        return sayBye;
};
let iWantToSayHi = letMeTalk( whatToSay: "hi");
iWantToSayHi("Sue");

let iWantToSayBye = letMeTalk( whatToSay: "bye");
iWantToSayBye("Lili");
```