

WEB230: JavaScript 1

Module 2: Functions

Defining a Function

- A variable that refers to a function
- `function` is a keyword
- Functions have a set of *parameters*, in this case only x

```
let square = function(x) {  
    return x * x;  
};  
  
let makeNoise = function() {  
    console.log("Pling!");  
};
```

Defining a Function Continued ...

- `square` has one *parameter*
- `makeNoise` has no *parameters*
- `square` produces a value
- `makeNoise` only has a *side effect*
- A `return` statement sets the returned value and exits the function

Bindings and Scopes

- Parameters behave like regular bindings (variables)
 - The value is set by the caller of the function
- Variables created inside a function are *local* to the function
 - This is referred to as *scope*
- Variables declared outside of any function are called *global*
 - They are visible throughout the program

Nested Scope

- Function definitions can include functions
- In this case, the scope can nest inside of another scope

Nested Scope Continued ...

```
const hummus = function(factor) {  
  const ingredient = function(amount, unit, name) {  
    let ingredientAmount = amount * factor;  
    if (ingredientAmount > 1) {  
      unit += "s";  
    }  
    console.log(`${ingredientAmount} ${unit} ${name}`);  
  };  
  ingredient(1, "can", "chickpeas");  
  ingredient(0.25, "cup", "tahini");  
  ingredient(0.25, "cup", "lemon juice");  
  ingredient(1, "clove", "garlic");  
  ingredient(2, "tablespoon", "olive oil");  
  ingredient(0.5, "teaspoon", "cumin");  
};
```

Functions as Values

- Function values can do all the things that other values do
 - use in expression
 - pass it as an argument to another function
- Variable that holds a function is still just a variable
 - can be redefined

Declaration Notation

- Shorter way to set a function
- Called a function *declaration*

```
console.log("The future says:", future());  
  
function future() {  
    return "We STILL have no flying cars.";  
}
```

- One subtle difference:
 - Function can be declared **below** the code that uses it

Arrow functions

- Third way of declaring functions
- Instead of the `function` keyword, it uses an arrow `=>`
- The arrow comes after the list of parameters and is followed by the function's body

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

Arrow functions Continued ...

- When there is only one parameter name, you can omit the parentheses around the parameter list
- If the body is a single expression then you can omit the braces and that expression will be returned

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

Optional Arguments

- You can call a function with too many or too few arguments
- Unneeded arguments are ignored
- Missing arguments are set to `undefined`
- You can test for missing arguments

Optional Arguments Continued ...

```
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
```

Growing Functions

- Sometimes you obviously need a function
- If a function name is easy to come up with it is probably a good case for a function
- Keep functions simple

Functions and Side Effects

- Two kinds of functions
 - Return a value
 - Have a side effect
- Avoid doing both in the same function
- *pure* functions
 - don't have side effects
 - don't use global variables that might change

