CPSC 1045: FUNCTIONS

Dr. Kim Lam

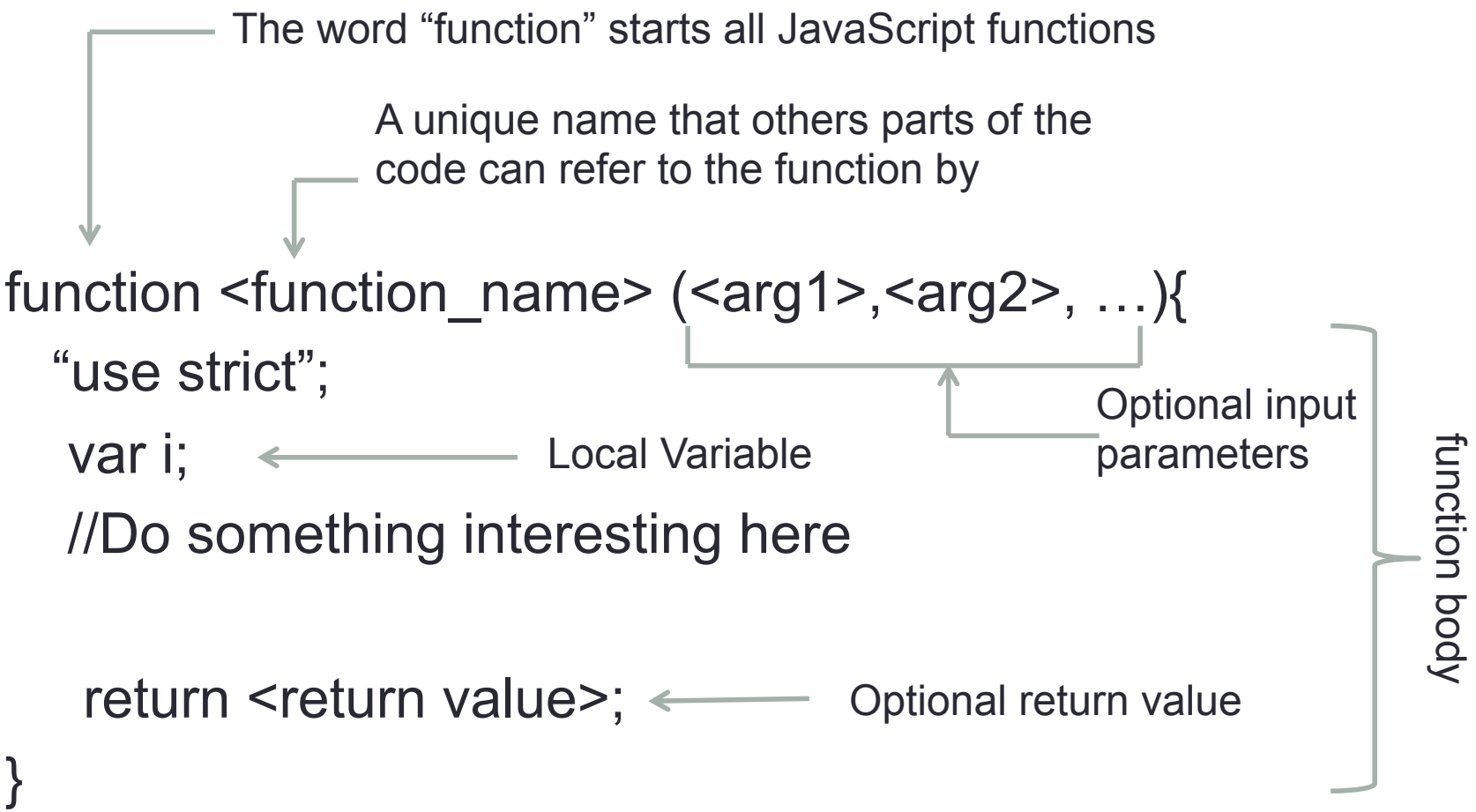
What are functions?

- Abstract Idea:
  - A function is a piece of code that you can **invoke/call** from an another section of code.
  - Our model of code running from top to bottom is even more broken.
  - Built In functions:
    - JavaScript has many built in functions, some of them are:
      - Math.abs(-1.0)
      - Math.floor(2.3)
  - When using functions written by others, you need to know what the function does, but not how it is implemented.
  - This allows people to share their work or work together in a convenient way, and each person can work on their own set of functions.

Why do we have a functions?

- Better program organization
  - Some function we only use once
  - But by giving sections of code meaningful names, our code is more readable.
- Easier to test
  - Can isolate small sections of code to test.
  - Can test them in the web console
- Reusing your code
  - Solving problems once and use the solution again!
- **Allows us to use the event driven model!**
- But…….
- It does take a little more effort to write programs using functions.
- Overall, you will save time by breaking down your program into smaller functions. The extra design time is well worth the effort.

Anatomy of a function



“use strict”;

- The first line of a function is **“use strict”;**
- It’s a string, followed by a semicolon
- In the JavaScript language this particular string at the **top** of a function has special meaning.

- This tells the JavaScript that we want to use **strict mode**.
- Not to worry we have already been doing this, but now we can have the interpreter tell us when we are not.
- Using this **strict mode**, reduces potentially dangerous and hard to debug errors.

When does a function run?

- Review:
  - JavaScript runs when our web-page, because we use the defer attribute.
  - <script src="example.js" defer></script>

  But When does a function run?
  - It does not run when they are defined
    - This is good, since we have used functions like **Math.random()** and we don't want them to run randomly.
  - They are run when they are invoked/called

  Example of invoking random and storing the resulting value:

  var temp = **Math.random();**

Example of a function

```
function sumFromOneToN(N){
    “use strict”;
    var temp = 0;
    var i;
    for(i = 1; i <= N; i = i+1){
        temp = temp + i;
    }
    return temp;
}
```

review: variables and the var keyword

- The **var** keyword is used to declare variables
- We should only declare variables once

```
var myVariable;
myVariable = " went over the sea ";
myVariable = myVariable + ". Went over the rainbow";

var hello = 886;
ctx.fillRect(hello, hello+100, 10,10);
```

Variable scope and functions

- Who can see your variables?
- Global/shared variables
  - Visible/modifiable by all functions
  - Variables declared outside a function can be seen by all functions.
  - All our variables so far were global.
- Local scope:
  - Only visible/modifiable by code inside the function
  - Variables declared inside a function can only be seen inside the function.
  - We should avoid naming local variables the same as global variables. If we do name them the same, local variables take precedence.

Example of variable scope

```
var fish = “fish for all”;

function store(item){
    “use strict”;
    var localBakery = “Knead for Dough”;
    var fishyReview = fish + “ is beside “ +
                      localBakery;
    return fishyReview;
}
```

- **fish** is a global variable
- **localBakery** and **fishyReview** are local variables
- **item** is a parameter and also a local variable
- “fish for all is beside Knead for Dough” will be the return value.

What happens when you invoke a function

1. JavaScript keep track of the position of where the function is being called, so it can return to that position later.
2. A copy of the parameters are created and the values filled in.
3. The functions starts running from the **top** of the function
    1. Until it reaches a **return** statement
    2. Or the **end** of the function
4. JavaScript figures out the return value
    1. It evaluates the expression after **return** and that is the return value
    2. If there is no expression after return then the value is the special value **undefined.**
    3. If it reaches the end of the function and there is no return statement, then the value is the special value **undefined.**
5. JavaScript then continues executing from where the function was called.

Side effects

- Side effect
  - Anything a function does that impact other functions

- How to produce side effects
  - Modifying global variables
  - Using shared objects and object properties
    - .innerHTML
    - .translate(), .rotate()
- Minimizing side effects
  - Ideally we should write code without side-effects
  - But this is GUI programming, so sometimes times the best we can do is minimize them.
  - for drawing functions, using **.save()** at the top of the function and **.restore()** at the end of the function.