CPSC 1045-Functions, variable scope, and program organization.

Functions

Functions are blocks of code that we grouped together and treat as a single unit. If we give the function a name, we can refer to it by that name. If we do not give it a name, it is known of an anonymous function.

We have to define a function before we can use them, otherwise JavaScript won't know what the function does. The step of using a functioning is known as **calling a function** or **invoking a function**.

An important thing to know is that:
Functions are not executed when they are defined; they are executed when they are called.

Why do we want to write functions

1. Helps organize our code
2. Allows code to be used more than once
3. Allows for easier testing
4. Allows for event driven programming

Using functions(review)

We have used many functions.

```
temp = Math.sin(0.5);

ctx.lineTo(10,10);
```

The function name followed by brackets is a **function call** expression. Like all expressions in JavaScript they evaluate to a value. The above two lines of code are examples of calling/invoking a function that we have used in the past. In the first example, `temp = Math.sin(0.5)`, we care about the value that `Math.sin(0.5)` and we store this value into the variable `temp`.

The second example, `ctx.lineTo(10,10),` we do not care about the value the function evaluates to, so we do not use it or store it anywhere. The reason we do not care about it's value is because the function has a side-effect. In this case, it adds

a point to CTX, the rendering context, and we use this function for it's side-effect and not the value it evaluates to.

How to define a function

One way to define a function:

```
function <functionName> ( ...parameters list ... ) {
  "use strict";
   //Function body

  return ...;

}
```

function: This is a JavaScript key word that we intend to define a function.
"use strict";: This specified that we are using the strict version of JavaScript. JSLint requires that all function follow the strict syntax.
<functionName>: If we want to give the function a name
(...parameters list ...): These are place holder variables that the function can use. When we call the function, then the variables in the parameter will have concrete values.
{ //Function body }: Code that is to be executed.
return ...: The value of the **function call** will be equal to the expression immediately after the statement.

```
function findGCD( num1, num2) {
  "use strict";
   var GCD = 1;
   for(var i =num2; i > 0; i = i -11){
       if( num1 % i === 0 && num2 % i === 0){
          GCD = i;
          break;
       }
   }
   return GCD;
}
```

findGCD: is the function name
num1, num2 : In this example we have two parameters num1 and num2. Together they are the parameter list.
return GCD: The value of the function will be the value stored in GCD, as the variable GCD is the expression after **return**.

When a program calls a function, the JavaScript interpreter will:

1. Fill in the parameters with the values the user provided
2. A new copy of all local variables are created
3. start running the function from the top
4. Stop when it reaches the end or a return statement.
5. The function call will evaluate to the value after the **return**
6. All local variables are destroyed
7. If no return is specified, or no return statement is specified the function call will evaluate to **undefined**.

Variable Scope

Variable scope refers to who can see a variable. We are interested in two levels of variables at this moment.

Global variables:

Global variables are available to everyone all the time and they are persistent. Any change made to a global variable is seen by everyone. Changes to global variables made inside a function is seen by other functions. Variables declared outside a function are global. All our variables so far have been global.

Local variables:

Local variables are those variables defined inside a function. Only code inside the function can access the copy of the local variables. A copy of the local variables is recreated every time the function is called. One call to a function uses a different set of local variables than a second call to the same function. Local variables are not shared across function calls.

If we leave out the `var` keyword when defining a variable, it is automatically global. When we use the `var` keyword when defining a variable inside a function then the only code inside the function can see the variable.

Side-Effects

Side effects are an action of a function call or line of code, which modifies the behavior of some other function call or line of code. Side effects are sometimes good and sometimes bad, but usually it's both.

The `ctx.translate` command is an example of a function that has a side effect of moving the local coordinate system for all subsequent drawing commands. For commands that want to work with the new local coordinate system this is a good thing. If our code is to work well with other code, this is a bad thing. So, we want to limit the side effects of `ctx.translate` to a small region of code. And example usage would be as follows.

```
function drawSqaure(x,y, ctx){
    "use strict";
    ctx.save();
    ctx.translate(x,y,);
    ctx.fillRect(0,0,100,100);
    ctx.restore();
}
```

The above function limits the effects of translate to only those lines before the restore statement. Because restore is the last statement in the function, drawing commands outside the function will not be affected by our `translate` command. In this way, we have limited the side-effect of translate only the lines of code we want it to effect. In fact we can have any number of `rotate` or `translate` commands between `save` and `restore`, and the side effects of those command will not leak out of the `save` and `restore` commands.

Anytime we modify some global or shared variable, we will cause side effects for those pieces of code that share the usage of the variable.

Writing cooperating functions with global/shared variables

There are two ways functions can work together.

1. A function can call another function. The caller then interacts with the callee through the return value and parameters.
2. Functions can cooperate with each other and communicate through shared variables. This is useful when one function ends and some code executes before the second function executes. This type of communication is **necessary** to make event driven programs work. This approach should **not** be used when it's not needed.

Two or more functions cooperate if there is a set of variables they all either modify or access. In our programs we accomplish this by using global variables, since they are visible to every function in our program. Only those functions who use the variables are cooperating. Cooperating functions, therefore inherently have side-effects. We have seen that the methods for the rendering context work in this way.

Example: Cooperating functions

```
var userAge;

function getInput() {
    "use strict";
    var inputString = "NaN";
    while (isNaN(Number(inputString))) {
        inputString = prompt("Enter your age please!");
    }
    userAge = Number(inputString);
}
```

```
function displayOutput(elementID) {
    "use strict";
    var ageDiv = document.getElementById(elementID);
    ageDiv.innerHTML = "Your age is " + userAge;

}

function setup(sectionId) {
    "use strict";
    getInput();
    displayOutput(sectionId);
}

setup("output");
```

In the above example **getInput()** and **displayoutput()** are said to be cooperating because they share the use of the **userAge** global variable. Neither function calls the other function. Both functions are run from **setup()**. For this example, it was not strictly necessary to structure code in this way, it was done to illustrate the concept of cooperation. When we move into event driven programming, the concept of cooperating functions will be absolutely necessary.