Our Flounder

Computer Science Correspondence School

discere domi

Lecture 13

CISC 131
Introduction to Programming and Problem Solving
Spring 2020
Programming Practices

Due: Tuesday, April 21, 2020, at start of class
Points: none

... but first a little story

The *Octave of Easter* is the eighth day after Easter Sunday and it was yesterday. Computer Scientists, as well as musicians, will recognize the letters *oct* as meaning eight. They are the root of the Latin *octo* which, of course, also means eight. Until the fourteenth century, nearly everything that was written in western Europe was written in Latin. It was, and continues to be, the official language of the Catholic church and, until the Second Ecumenical Council of the Vatican (Vatican II) in 1962, Latin was the language for all its rituals including the Mass and its liturgical book, the missal.

Latin was certainly used in the Mass in 1831 when Victor Hugo wrote the *Hunchback of Notre Dame*, which, I'm guessing you read as part of a high school English class. You may remember that in the novel, the archdeacon of Notre Dame comes upon the abandonded baby and names him Quasimodo. That happened on the Octave of Easter. The Introit is a part of the Mass and, on the Octave of Easter, it's opening line in English is something like *as newborn babes*. You might find the Latin version a bit more interesting: *Quasi modo geniti infantes*. Now you know and you can impress your friends.

General Comments about Programming

There often many ways to write a function that will solve a problem. Some are better than others and, often, I will require you to implement a function using constraints in order to give you some practice with the required techniques. The simple date functions that we wrote some time ago are good examples of this. These functions:

```
getDay   ( yyyymmdd )
getMonth ( yyyymmdd )
getYear  ( yyyymmdd )

setDay   ( yyyymmdd )
setMonth ( yyyymmdd )
setYear  ( yyyymmdd )
```

all take an integer parameter and return an integer result. Their implementation should only do numeric processing. There should be no *String* processing done in any of them. For example, this function is more complicated than it needs to be:

```
function getDayString( yyyymmdd )
{
 var result;
 result = "" + yyyymmdd;
 result = result.substring(result.length-2)
 return result;
}
```

Because the parameter is an integer, ut would be far better to use numeric procesing:

```
function getDay   ( yyyymmdd ) { return yyyymmdd % 100; }
```

Step 1:
Check your *Dates.js* library and change any *String* implementations of these functions into purely numeric implementations.

```
getDay   ( yyyymmdd )
getMonth ( yyyymmdd )
getYear  ( yyyymmdd )

setDay   ( yyyymmdd )
setMonth ( yyyymmdd )
setYear  ( yyyymmdd )
```

Step 2:
Relational operators such as ===, <, >, and others compare two values and evaluate to *true* or *false*. It is, generally, poor programming practice to have an *if-the-else* statement assign *true* or *false* to a variable. For example, this is not a well-written function:

```
function demo ( x )
{
 var result;
 if ( x > 6 )
  result = true;
  else
  result = false;
 return result;
}
```

This function should be written this way:

```
function demo ( x ) { return x > 6; }
```

Here is a good exmple of testing for a valid year in a date. It does not test the variable and store a Boolean result. It simple returns the Boolean result. This is a much better approach.

```
function isValidYear ( yyyymmdd ) { return getYear(yyyymmdd) !== 0; }
```

Check your *Dates.js* library and change any implementations that can use this better approach.

Step 3:
Each function should have a single *return* statement. You should structure your code in such a way that this is possible. You will see many, many examples of poor programming practice on the internet. The rule for our class is that each function may have only a single *return* statement. Check your *Dates.js* library and change any implementations that use multiple *return* statements into implementations that use a single *return* statement.

Step 4:
Simplify your implementations. If you are processing numbers, think how the numbers relate to one another. For example, the *getEarlierDate* function can be written clearly in a few lines:

```
function getEarlierDate( yyyymmdd1, yyyymmdd2 )
{
 var earlier;
 if ( yyyymmdd1 < yyyyymmdd2 )
  earlier = yyyymmdd1;
  else
  earlier = yyyymmdd2;
 return earlier;
}
```

or even a single line:

```
function getEarlierDate( yyyymmdd1, yyyymmdd2 )
{
 return Math.min ( yyyymmdd1, yyyymmdd2 );
}
```

Check your *Dates.js* library and simplify them if possible.

Solutions to Problems from Lecture 11

Study the implementations shown below for the homework problems. Make sure you understand how each one works

`function getEarlierDate ( yyyymmdd1, yyyymmdd2 )`

This function is passed two dates in the *yymmdd* format and returns the one that is earlier in time. If both dates are the same, either one can be returned because they are the same.

```
function getEarlierDate( yyyymmdd1, yyyymmdd2 )
{
 return Math.min ( yyyymmdd1, yyyymmdd2 );
}
```

`function getLaterDate   ( yyyymmdd1, yyyymmdd2`

This function is passed two dates in the *yymmdd* format and returns the one that is later in time. If both dates are the same, either one can be returned because they are the same.

```
function getLaterDate   ( yyyymmdd1, yyyymmdd2 )
 {
  return Math.max ( yyyymmdd1, yyyymmdd2 );
}
```

`function sameDate ( yyyymmdd1, yyyymmdd2`

This function returns Boolean *true* if the parameters are the same date and *false* if they are not the same date.

```
function sameDate ( yyyymmdd1, yyyymmdd2 ) { return yyyymmdd1 === yyyymmdd2; }
```

`function sameYear  ( yyyymmdd1, yyyymmdd2`

This function returns Boolean *true* if the two parameter dates have the same year value and *false* if they do not have the same year value.

```
function sameYear  ( yyyymmdd1, yyyymmdd2 )
 {
  return getYear(yyyymmdd1)  === getYear(yyyymmdd2);
 }
```

`function sameMonth ( yyyymmdd1, yyyymmdd2 )`

This function returns Boolean *true* if the two parameter dates have the same month value and *false* if they do not have the same month value.

```
function sameMonth ( yyyymmdd1, yyyymmdd2 )
{
 return getMonth(yyyymmdd1) === getMonth(yyyymmdd2);
}
```

`function sameDay   ( yyyymmdd1, yyyymmdd2 )`

This function returns Boolean *true* if the two parameter dates have the same day value and *false* if they do not have the same day value.

```
function sameDay   ( yyyymmdd1, yyyymmdd2 )
{
 return getDay(yyyymmdd1)   === getDay(yyyymmdd2);
}
```

`function getDayName ( dayNumber )`

This function is passed an integer value. If the value is between one and seven, it returns a *String* containing the name of the week associated with that value. *Sunday* is associated with value one, *Monday* is associated with value two, and so on, with *Saturday* being associated with value seven. If the value is not between one and seven, the function returns a zero length *String*.

You must use an array to store and access the names of the days. This function can be implemented without any *if* statements but you may use at most one *if* statement.

Here are a couple of good implementations, one that uses an *if* statement and one that does not.

```
function getDayName ( dayNumber )
{
 var dayName;
 var result;

 dayName = [
            "Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday"
           ];

 result = "";
 if ( dayNumber >=1 && dayNumber <= 7)
 {
  result = dayName [ dayNumber - 1 ];
 }

 return result;
}
```

```
function getDayName ( dayNumber )
{
var dayName;

 dayName = [
            "",
            "Sunday", "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday",
            ""
           ];
 dayNumber = Math.max ( 0, dayNumber );
 dayNumber = Math.min ( dayNumber, 8 );
 return dayName [ dayNumber ];
}
```

`function getMonthName ( monthNumber )`

This function is passed an integer value. If the value is between one and twelve, it returns a *String* containing the name of the month associated with that value. *January* is associated with value one, *February* is associated with value two, and so on, with *December* being associated with value twelve. If the value is not between one and twelve, the function returns a zero length *String*.

You must use an array to store and access the names of the months. This function can be implemented without any *if* statements but you may use at most one *if* statement.

This function is nearly identical to the getDayName function. The only differences are the array that is used and the range of values in the parameter

Step 5:

After you have applied the approaches in Steps 1 through 4, add these functions to your *Dates.js* library:

```
function getEarlierDate ( yyyymmdd1, yyyymmdd2 )
function getLaterDate   ( yyyymmdd1, yyyymmdd2
function sameDate ( yyyymmdd1, yyyymmdd2
function sameYear  ( yyyymmdd1, yyyymmdd2
function sameMonth ( yyyymmdd1, yyyymmdd2 )
function sameDay   ( yyyymmdd1, yyyymmdd2 )
function getDayName ( dayNumber )     Array implementation only
function getMonthName ( monthNumber ) Array implementation only
```

Start of Today's Lecture

Testing is an important part of programming. It is the most important part. Without thorough testing, there is no way to know whether the functions you have written work correctly or not. Even the simplest functions should be thoroughly tested.

Testing early is an important part of programming. Writing too much code before you test can result in many wasted hours of debugging. When you find a problem, try these steps to help identify it cause.

- Keep the Firefox error console open and check it for errors. It will tell you the line number where it encountered the error. The actual error will be on or before that line number.
- Check for typos.
- Remember that only way to change the value of a variable is to use an assignment statement. Formal parameters are initialized using the actual parameter values and that happens automatically when the function is called. All other variables require you to use an assignment statement. For example:
```
Math.min (x, y );  // this statement does nothing useful
z = Math.min(x,y); // this statement changes the value of z
```
- If the function is designed to have formal parameters, have you listed the formal parameters in the function header?
- If the function is designed to have formal parameters, have you passed corresponding actual parameters to the function when it is called? Have you passed the actual parameters in the correct order such that they match up with the corresponding formal parameters? If you forget to pass an actual parameter, the formal parameter with which it is associated will have the value *undefined*. If you forget to initialize a variable, its initial value will be *undefined*, as well.
- If the function was designed to return a value, is the last statement of your function the word *return* followed by a variable name or expression? Are you returning the correct variable or expression?
- If you call a function that returns a value, have you stored the returned value or used it in an expression? If a function that returns a value is called without storing the return value or using it in an expression, the returned value is simply thrown away.
- Make sure you include the *"use strict"* directive at the beginning of each Javascipt program. It will cause the interpreter to warn you about undeclared (sometimes mistyped) variables when the Firefox error console is open.
- Make sure any loop meets the two requirements for writing correct loops: initialization and change.
- Make sure that you do not have a semicolon following the loop statement. For example,
```
while(i<9);
```
is an infinite loop and
```
for(i=0; i<source.length; i=i+1);
```
is a loop that has no statements in the loop body.
- Make sure that you do not have a semicolon following the *if* statement. For example,
```
if(i<9);
```
won't do anything.

- If a loop isn’t working correctly, put a *window.alert* statement in your loop to show you the value of the variables that are being used in the loop. This may help you figure out what is going wrong.
- Don’t confuse a function reference with a function call. Assigning a function to an event (such as *onclick*) must be done using a function reference, not a function call.