Birkbeck College, University of London
School of Computer Science and Information Systems

Programming Building Blocks

George Roussos and Constantinos A. Constantinides

TABLE OF CONTENTS

INTRODUCTION

This is a set of notes for students of IT Applications and Foundations degree in IT. These notes have been prepared as to cover some aspects of software development that usually are omitted from the modules due to time pressures. Rather than looking in the specifics of languages, web technologies and development tools, we will look at the main skills that programmers must develop to be able to use their knowledge effectively for tangible results.

Thus, we will look at the mental skills of analysis and organization of work to turn the knowledge acquired in ITApps to produce results.

1. Introduction to problem solving

An *algorithm* is a sequence of simple steps that can be followed to solve a problem. These steps must be organized in a logical, and clear manner.

We design algorithms using three basic methods of control: sequence, selection, and repetition.

1.1 Sequential control.
Sequential Control means that the steps of an algorithm are carried out in a sequential manner where each step is executed exactly once.

Let's look at the following problem: We need to obtain the temperature expressed in Farenheit degrees and convert it to degrees Celcius. An algorithm to solve this problem would be:

1. Read temperature in Farenheit
2. Apply conversion formula
3. Display result in degrees Celcius

In this example, the algorithm consists of three steps. Also, note that the description of the algorithm is done using a so-called *pseudocode*. A pseudocode is a mixture of English (or any other human language), symbols, and selected features commonly used in programming languages. Here is the above algorithm written in pseudocode:

```
READ degrees_Farenheit
degrees_Celcius = (5/9)*(degrees_Farenheit-32)
DISPLAY degrees_Celcius
```

Another option for describing an algorithm is to use a graphical representation in addition to, or in place of pseudocode. The most common graphical representation of an algorithm is the *flowchart*.

1.2 Selection control.
In *Selection Control* only one of a number of alternative steps is executed. Let's see how this works in specific examples.

<u>Example 1</u>. This is an algorithm that decides whether a student should receive a distinction.

```
READ grade
IF (grade >= 95)
     THEN student receives a distinction
```

This is the simplest case of selection control. In fact, in essence it is a sequential form but the second step is only taken when the condition contained in the brackets is true, that is the last step (starting at THEN) is *selected for execution* only when the condition is satisfied.

Example 2. Control access to a computer depending on whether the user has supplied a username and password that are contained in the system database.

```
IF (username in database AND password corresponds
to user) THEN
       Accept user
ELSE
       Deny user
```

In this case, there is also a decision to be made but a more complex one. In the condition is true only the `Accept user` step is executed and if the condition is false only the `Deny user` step is executed. That is, the *condition* helps us make a *selection* which *controls* which of the two steps will be executed.

Now look more closely to the condition. Does it guarantees that the user is valid? What if we had this condition instead, which looks pretty similar: `IF (username in database AND password in database)`? What would happen is somebody typed a valid username but the password of a different but also valid user? Does this second condition satisfies our requirement that to access this account one should be the right user and know the corresponding password?

When you write a condition always make sure that it performs exactly the check that you intended it to! It is very easy to translate words in pseudocode that does a different thing than what was intended!

Example 3. Decide on a student's grade based on their coursework mark.

```
READ result
CASE (result >=90)
  PRINT 'A'
CASE (result >=80)
  PRINT 'B'
CASE (result >=70)
  PRINT 'C'
CASE (result >=60)
  PRINT 'D'
CASE (result <60)
  PRINT 'F'
```

In this case, there are five conditions which are checked, one after the other.

Question. What is the difference between examples 1, and 2, and 3?
Answer. There are three differences:

1. Example 1 provides no alternative action is the condition is not true.
2. Example 2 provides an alternative action when the conditions are false.
3. Example 3 provides multiple alternatives to select from.

1.3 Repetition.

In *Repetition* one or more steps are performed repeatedly.

Example 1. Compute the average of ten numbers:

```
total = 0
average = 0
FOR 1 to 10
  Read number
  total = total + number
ENDFOR
average = total / 10
```

Example 2. Read numbers and add them up until their total value reaches (or exceeds) a set value represented by *S*.

```
WHILE (total < S) DO
       Read number
       total = total + number
ENDDO
```

Question. What is the difference between Examples 1, and 2?
Answer. The difference is that in Example 1 we know beforehand the exact number of repetitions that will be carried out (they are 10), but in Example 2 we do not know beforehand (it depends on the particular numbers that will be fed to our program during its execution).

Sequence, selection, and repetition are by themselves simple. But put together, they can construct any algorithm that we can imagine.

2. SOFTWARE DEVELOPMENT

A *program* is a sequence of instructions that a computer follows to solve a problem. A program is written in a *computer language*. Most of the languages have been created to fit particular classes of problems. *Javascript* is a programming language that has been created to support *dynamic web pages* (rather than and static pages that are created using HTML) and is by the most widely used language in this area. The process of collecting requirements, analyzing a problem and designing its solution as well as writing and maintaining a computer program is called *software development*.

2.1 The software engineering method for problem solving

The *software engineering method* is a way to approach problem solving using a computer program and has the following five steps:

1. *Specify the problem requirements*. Describe the problem completely and unambiguously.
2. *Analyze the problem* requirements. Identify (a) inputs, (b) the data to work with, (c) outputs (i.e. desired results).
3. *Design the algorithm to solve the problem*. Write the step-by-step procedures required to solve the problem. For large problems, don't attempt to solve every last detail of the problem at the beginning. Use a *top-down design* that is, list the major steps first and break down the problem into smaller and more manageable sub-problems. Once you know the sub-problems you can attack each one individually. That might require breaking down a sub-problem into even smaller sub-problems. This is called *algorithm refinement*. This process of breaking down a problem into its sub-problems is also called *divide-and-conquer*.
4. *Implement the algorithm*, that is convert the algorithm into a program.
5. *Test and verify the completed program*. Test the completed program to verify that it works as desired. Do not rely on one test case. Run the program several times using different sets of data. Try to imagine what somebody who would like to violate your program would try to do.

Let's see how this method works with a specific example:

1. Problem statement. You are the receiving clerk for an electronics parts distributor that imports large quantities of wire and cable produced abroad and sold in the UK. Most of the wire you receive is measured in meters; however, your customers order wire by foot. Your supervisor wants you to compute the length in feet of each roll of wire or cable that you stock. You have to write a program to perform this conversion.

2. Analysis. You must convert from one system of measurement to another. The problem statement says that you receive wire measured in meters; thus, the problem input is wire length in meters. Your supervisor wants to know the equivalent amount in feet, which is your problem output.

Next, you need to know the relationship between meters and feet. If you do not know this already you open up a book that explains how you can convert length measurements (for example B. Franco, Key to Metric Measurement, Emeryville, CA.: Key Curriculum Press, 2000) or you search on the Web (for example `http://www.convertit.com/Go/ConvertIt/Measurement/Converter.ASP` -- but be careful to find a trustworthy site!). For example, if you look it up in a table of conversions factors, you will find that one meter equals 39.37 inches. We also know that there are 12 inches to a foot. These findings are summarized below:

Inputs:	Wire length in meter (wmeter)
Outputs:	Wire length in feet (wfeet)
Additional program variable:	Wire length in inches (winchs)
Relevant formulas or relations:	1 meter = 39.37 inches 1 foot = 12 inches

3. Design. Next, we have to formulate the algorithm to solve the problem. We begin by listing the three major steps, or sub-problems, of the algorithm:

1. Read the wire length in meters
2. Convert the wire length from meters to feet
3. Display the wire length in feet

We then must decide whether any steps of the algorithm need further refinement. Steps 1, and 3 need no further refinement. The refinement of step 2 shows how to perform the conversion:

2.1 Convert the wire length from meters to inches
2.2 Convert the wire length from inches to feet.

The complete algorithm with refinements is shown below:

```
1. Read the wire length in meters
2. Convert the wire length from meters to feet
    2.1 Convert the wire length from meters to inches
    2.2 Convert the wire length from inches to feet.
3. Display the wire length in feet
```

4. Implementation. The next step is to implement the algorithm as a Javascript function.

```
/*
  Converts wire length from meters to feet.
  The user enters the wire length in meters.
*/

function convert(wmeter)
{
// Helper variables
var inchmt = 39.37
var inchft = 12
var wfeet=0
var winchs=0

// Convert the wire length from meters to feet
winchs = wmeter * inchmt
wfeet = winchs / inchft

// Return the wire length in feet
return wfeet
}
```

We can now use this function in a Web page and use it to convert meters to feet:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
     "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<script type="text/javascript">
/*
  Converts wire length from meters to feet.
  The user enters the wire length in meters.
*/
function convert(wmeter)
{
// Helper variables
var inchmt = 39.37
var inchft = 12
var wfeet=0
var winchs=0
// Convert the wire length from meters to feet
winchs = wmeter * inchmt;
wfeet = winchs / inchft;
// Return the wire length in feet
return wfeet
}
</script>

</head>
<body>
<script type="text/javascript">
document.write("Length in feet is: ",convert(10));
</script>
</body>
```

A sample output of the program is shown below:

```
Length in feet is: 32.80833333333333
```

Let's look at the different elements that we used to create this program.

Variables. Variables are quantities that can retain any value each time a program runs:

```
var inchmt
```

Also `wfeet`, and `inchft` are variables. It is very important that variables have meaningful names, because they make program easier to read, and understand. If, for example, you name your variables `m` and `f`, it would not be straightforward to remember what they represent.

Each language has special rules that restrict your selection of variable names. In Javascript a variable name can be made up of small or capital letters, digits, and underscore characters and the first character must always be a letter or the underscore. Other characters, such as blanks, must not appear within a name.

A variable can be initialized (assigned to a value) during declaration for example

```
var inchmt = 39.37
```

or it can be assigned a value later on (after its declaration), such as

```
winchs = wmeter * inchmt;
```

Types. Variable `inchmt` holds a real value, and `inchft` holds an integer value. Reals contain a fractional part, whereas integers are whole numbers.

Comments. Any line that starts with `//` is taken as a comment and thus ignored by the computer. A comment can also be anything contain between `/*` and `*/` for example:

```
/*
  Converts wire length from meters to feet.
  The user enters the wire length in meters.
*/
```

and

```
// Helper variables
```

Comments are very important! They give out information about the program to you and they will come very handy when you need to change your code a few months after creating it. Other people who have to work with your code will also benefit. You should always use comments in your programs to explain anything that is not straightforward to understand.

Assignment statements. We use assignment statements to give a value to a variable. The assignment statement

```
winchs = wmeter * inchmt;
```

assigns the value of the expression `wmeter * inchmt` to variable `winchs`.

Operations. Addition and subtraction are denoted by the usual + and - symbols. Multiplication is denoted by `*`, and division by `/`. The order of evaluation follows the corresponding mathematical rules. Parentheses have higher precedence in an expression. Multiplication and division follows. Addition, and subtraction have the lowest precedence. Operations of the same precedence are executed from left to right.

5. Testing and verification. Compare the program results with those that you compute manually or by using a calculator. Keep in mind that programs often have errors. We distinguish between two types of errors: Syntactic (grammatical), and semantic (logical) errors.

Syntactic errors are caused by misspelled words, or the omission of a word or part of it, such as

```
docment.write("Length in feet is: ",convert(10));
```

Semantic errors indicate that there is an error in the logic of the program. Some semantic errors might cause a failure (e.g. divide by zero), others might just provide wrong results. For example, if we had used

```
winchs = wmeter + inchmt;
```

instead of the correct

```
winchs = wmeter * inchmt;
```

in the example above, then everything would seem to work as required but the calculated result would be incorrect.

3. Using Functions for Modular Programming

In the previous section we discussed the divide-and-conquer approach to program desing. The main idea was to divide the overall task in smaller chunks that we can handle more easily. Sometimes there are tasks that are being repeated in several points in the program – we would want to write code only once!

To do this in Javascript, we use *functions*. Functions are relatively small program units designed to perform a particular task. Functions have other characteristics as well: When they terminate their execution, they return a single value; they return values via their names; and finally, they are referenced by using their name in an expression. For example, in the example of the previous section we called functions `convert` in the following statement:

```
document.write("Length in feet is: ",convert(10));
```

There are mainly two different types of functions:

Library functions. Javascript provides many *built-in or library functions*. Any of these functions can be used in any expression by giving its name followed by the actual arguments to which the function is to apply.

For example, if we want to find the smaller of two variables we can and uses `Math.min` function:

```
var num1, num2
smaller=Math.min(num1,num2);
```

Function MIN gets three arguments: num1, num2, and num3, and returns the minimum of the arguments that is assigned to the variable minimum

Function subprograms. In many cases it is best to define additional functions. A *programmer-defined function* can be used in the same way as any library function. The general form of a programmer-defined function is

```
function myfunction(argument1,argument2,etc)
{
some statements
}
```

Example: Let's write a function that calculates the area of a square given the length of its side.

```
function square_area(side)
```

```
{
return side*side
}
```

Functions that will return a result must use the `return` statement. This statement specifies the value that will be returned to the point where the function was called.

Functions as an aid to program structure

There are several advantages of using functions:

- They enable the programmer to break down the design of the program into several smaller sections, thus making the analysis and programming task easier.
- They enable the programmer to write and test each of the smaller sections independently of the rest of the program.
- In a large project different persons independently of each other can develop these smaller components.

4. Structured Programming: Sequential Structure

In section one we learnt that any algorithm (and thus any program) can be structured using the three basic control structures: *sequence, selection, and repetition.* In this and the following sections we will look at each of these structures in turn, starting with the simplest one, the sequential structure.

The sequential structure is a simple way to refer to the execution of a sequence of statements in the order in which they appear so that each statement is executed exactly once. Consider the following sequence of statements:

```
var argv = SetCookie.arguments;
var argc = SetCookie.arguments.length;
var expires = (2 < argc) ? argv[2] : null;
var path = (3 < argc) ? argv[3] : null;
var domain = (4 < argc) ? argv[4] : null;
var secure = (5 < argc) ? argv[5] : false;
document.cookie = name + "=" expires.toGMTString();
```

When executed, this sequence creates a cookie.

4. STRUCTURED PROGRAMMING: SELECTION STRUCTURE

The selection structure can take different forms: single selection, nested ifs and multiple selection. We will discuss each case individually.

Single Selection

In the simplest selection structure, a sequence of statements (also called a block of statements) is executed or bypassed depending on whether a given logical expression is true or false. In Javascript the general form of the simple selection is:

```
if (logical expression is true) {
  statement
}
```

It is also possible to specify an alternative statement sequence for execution when the logical expression if false. The general form of the simple selection with alternative action is:

```
if (logical expression is true) {
  statement
} else {
  some other statement
}
```

Example: The following code fragment reads the time from the system clock and if it is earlier than 10am it displays the message "Good morning!"

```
var d = new Date()
var time = d.getHours()

if (time < 10)
{
document.write("<b>Good morning!</b>")
}
```

Example: The following code fragment reads the time from the system clock and if it is earlier than 10am it displays the message "Good morning!" and in every other case it prints "Good day!".

```
var d = new Date()
var time = d.getHours()

if (time < 10)
```

```
{
document.write("<b>Good morning!</b>")
}
else
{
document.write("<b>Good day!</b>")
}
```

Nested IF Statement

The statements in an IF construct may themselves contain other IF constructs. In this case, the second IF construct is said to be *nested* within the first.

Example: Consider the following program that displays a student's grade:

```
function  display_result(mark) {
if (mark >= 90) {
       display.document("A");
} else if (mark >= 80) {
       display.document("B");
} else if (mark >= 70) {
       display.document("C");
} else if (mark >= 60) {
       display.document("D");
} else {
       display.document("F");
}
}
```

The order in which the choices are considered is very important!

Example: Consider the following program that computes the roots of a quadratic equation:

```
function quadratic(a, b, c) {
// program to compute the roots of a quadratic equation
var  root1, root2, disc

// compute discriminant
disc = b * b - (4.0 * a * c);

// consider all cases for discriminant
if (a == 0) {
  root1 = -c/b;
  display.document("Equation is not quadratic");
  display.document("The single real root is ",root1);
} else if (disc > 0) {
  root1 = (-b + SQRT(disc)) / (2.0 * a);
```

```
          root2 = (-b - SQRT(disc)) / (2.0 * a);
          display.document("The two real roots are: ",/
               root1,   root2);
        } else if (disc == 0) {
          root1 = -b / (2.0 * a);
          display.document("There is only one real root");
          display.document("The root is ", root1);
        } else {
          display.document("There are two complex roots");
        }
        }
```

Multiple Selection

The selection structure considered thus far, involved selecting one of two alternatives. It is also possible to use the IF construct to design selection structures that contain more than two alternatives. The `switch` construct is used to select among multiple alternatives on the basis of a single expression value. The general form of the case structure is

```
switch (expression)
{
case label1:
  code to be executed if expression = label1
  break
case label2:
  code to be executed if expression = label2
  break
default:
  code to be executed
  if expression is different
  from both label1 and label2
}
```

The expression is evaluated and compared to each selector, where each selector is a list of possible values of the expression. Only one statement is executed. The DEFAULT statement is **optional** and it is obeyed if none of the other CASE statements produces a match.

Example: The following code fragment will return a different response depending on the day it is run:

```
var d = new Date()
theDay=d.getDay()
switch (theDay)
{
   case 5:
     document.write("Finally Friday")
   break
   case 6:
     document.write("Super Saturday")
   break
   case 0:
     document.write("Sleepy Sunday")
   break
   default:
     document.write("I'm really looking forward to this
weekend!")
}
```

Comparison between a nested IF and the case structure

We can start with an example: Consider the following price structure for concert tickets:

Section	**Ticket price**
0-99	
100-199	25.00
200-299	25.00
300-399	20.00
400-499	15.00

Using a nested IF statement, one can have the following structure to assign the desired value to price or displays an error message if the section is not listed in the table:

```
if (section == 50) {
  price = 50.00;
} else if   ((100 <= section) &&(section <= 199)) ||
  (200 <= section) && (section <= 299)) {
  price = 25.00;
} else if   ((300 <= section) && (section <= 399)) {
  price = 20.00;
} else if   ((400 <= section) && (section <= 499)) {
  price = 15.00;
} else {
  document.write("Invalid section number”);
}
```

Using the `switch` statement, we can rewrite the above program in the following way:

```
switch (Math.floor(section/100)){
case 0:
   price = 50.00;
   break;
case 1:
   price = 25.00;
   break;
case 2:
   price = 25.00;
   break;
case 3:
   price = 20.00;
   break;
case 4:
   price = 15.00;
   break;
default:
   document.write("Invalid section number");
}
```

Clearly, the case structure is more readable, and easier to code than the nested IF structure.

6. STRUCTURED PROGRAMMING: REPETITION STRUCTURE

The repetition of a block of statements a number of times is called a *loop*, and is so important that Javascript support several different constructs to support this feature.

The FOR construct

The FOR statement repeats a block of statements a predetermined number of times, as specified by the initial value of an index variable that is incremented until it reaches some final value. The general form of the do statement is

```
for (initialization; condition; increment)
{
    code to be executed
}
```

In this case, the index variable is initialized at `initialization` and the `code` is executed until `condition` is reached. At the end of each iteration the variable is increased by `increment`.

<u>Example 1</u>: The following program segment will display the first 10 positive integers:

```
for (i = 1; i <= 10; i++)
{
   document.write(" ", i)
}
```

The output will be: `1 2 3 4 5 6 7 8 9 10`

<u>Example 2</u>: In the following example, the *initial value* for variable `i` is 1, and it is incremented by 2, so the next values will be 3, 5, and so on. The following segment would display the odd positive integers between 1 and 10.

```
for (i = 1; i <= 10; i+=2)
{
   document.write(" ", i)
}
```

Output will be: `1 3 5 7 9`

In this example, when the `i` variable exceeds its final value (i.e. 10), there are no more repetitions.

The following segment will compute the average of the first ten numbers:

```
var limit = 10                          // holds number of repetitions
var index                               // holds initial value
var average                             // holds the average
var total = 0                           // holds sum; initialized to 0

for (index = 1; index<=limit; index++) {
   total = total + index                // compute new total
}
average = total / limit                 // compute average
```

The WHILE construct

The `while` statement will execute a block of code while a `condition` is true.

```
while (condition)
{
    code to be executed
}
```

Example 3. Let's redo Example 1 using the `while` construct

```
i = 0
while (i <= 5)
{
   document.write(" ", i)
   i++
}
```

The DO…WHILE construct

The `do...while` statement will execute a block of code once, and then it will repeat the loop while a `condition` is true

```
do
{
    code to be executed
}
while (condition)
```

Example 4. Let's redo Example 1 using the `do..while` construct

```
i = 0
do
{
document.write(" ", i)
i++
}
while (i <= 5)
```

Summary

Very often when you write code, you want the same block of code to run a number of times. You can use looping statements in your code to do this. In JavaScript we have the following looping statements:

- **while** - loops through a block of code while a condition is true
- **do...while** - loops through a block of code once, and then repeats the loop while a condition is true
- **for** - run statements a specified number of times