

*David M. Keil*  
*Framingham State University*

# 6. Problem solving and programming

1. Specification and design
2. Algorithm design tools
3. The loop control structure
4. JavaScript and event-driven software

*Reading:*  
Ch. 10;  
Handouts

David Keil Introduction to Information Technology 1/12 1

## Inquiry

- Why does almost everyone either hate or love programming?
- Why does software not work?
- What steps and what tools enable building computer solutions efficiently?

David Keil Introduction to Information Technology 1/12 2

## Systems and system development

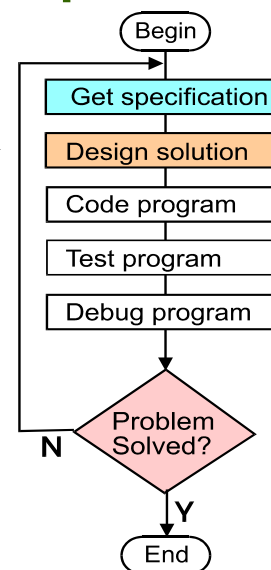
- *System*: a group of components working together toward a goal
- *Information system*: hardware, software, staff, procedures, ways to generate and store data
- A *systems development life cycle* describes steps in analysis, design, implementation, testing, and maintenance
- The job categories of *systems analyst* and *software engineer* carry out these tasks

## The system development process

The development process is iterative

*Phases:*

- *Analysis*: specifies desired input and output
- *Program design*: prepares algorithms, plans
- *Coding*: implements design by writing a *program* in a language
- *Testing*: evaluates working program
- *Maintenance*: addresses errors and needs not found in previous stages



### Objectives

- 6a. Identify the steps in system development
- 6b. Explain the notion of an algorithm
- 6c. Trace a looping and branching computation specified in a flowchart
- 6d. Write a looping flowchart or pseudocode
- 6e. Write branching and looping code in a procedural language
- 6f. Explain the concept of debugging
- 6g. Create a simple event-driven web page

### 1. Specification and design

*Properties of system specifications:*

- Input
- Output
- Relationship between input and output
- Definiteness (deterministic sequence of operations)
- Effectiveness (doability)
- Finiteness and reasonableness of time

*Designs tell how to satisfy the specifications*

### An example problem description

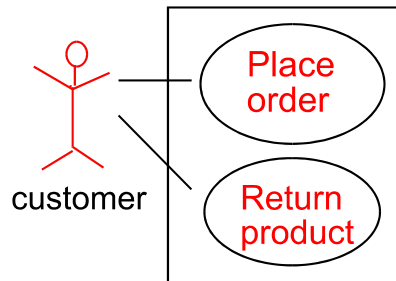
- What service must the software provide?  
*Example: Process customer orders from catalog*
- What assumptions are made?  
*Example: Some customers may find Web access convenient*
- What risks are involved?  
*Example: Some users are inexperienced with Web access*

### Unified Modeling Language

- A standard graphical notation for system specification and design
- Motivated by need to depict interactions between systems and their environments, initiated by external *actors*
- Diagrams include *use-case, activity, class, state, interaction*
- Supports an *object-oriented* methodology

## Use cases and system specification

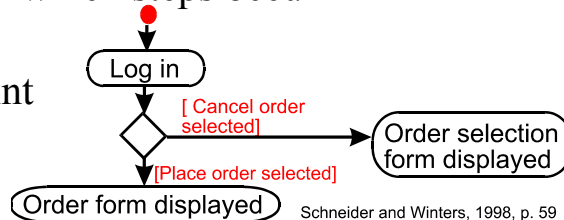
- *Use case*: A typical interaction between system and an *actor* in its environment
- *Actor*: A role, such as customer, manager, supplier, salesperson
- Actors initiate use cases
- Example of two uses cases (UML use-case diagram):



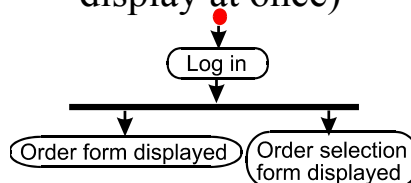
## UML activity diagrams

- Depict order in which steps occur

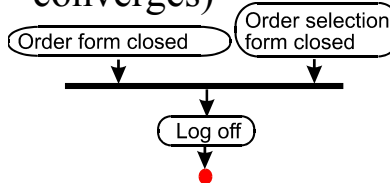
- *Examples:*  
A decision point (branching)



- A *fork* (2 forms display at once)



- A *join* (execution converges)

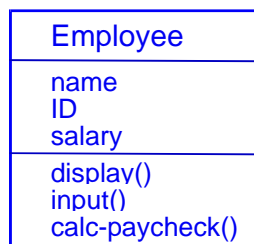


## The object-oriented paradigm

- *Object*: An instance of a class, with attributes
- *Class*: A category defined by data *attributes* (properties) and *operations* (“methods”)
- In object-oriented programming, objects interact with each other via *messages*
- Objects may *contain* other objects
- Subclasses may *inherit* from other classes
- *Example*: Right-clicking an icon displays the properties and methods of the class of the object the icon represents

## UML class diagrams

- Rectangle with 3 horizontal compartments



- UML class association diagram

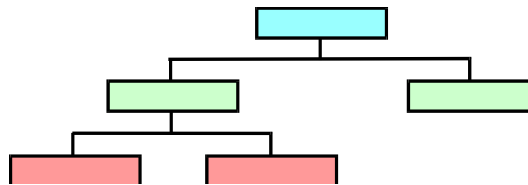


## Systems design

- *Solution design is:*
  - Based on specification (analysis)
  - Language independent
- *Tools:* flowcharts, pseudocode, module hierarchies, Unified Modeling Language diagrams
- Coding, testing come after design

## Modular design

- One strategy: divide and conquer
- All languages use modularity
- A hierarchical organization may be modular

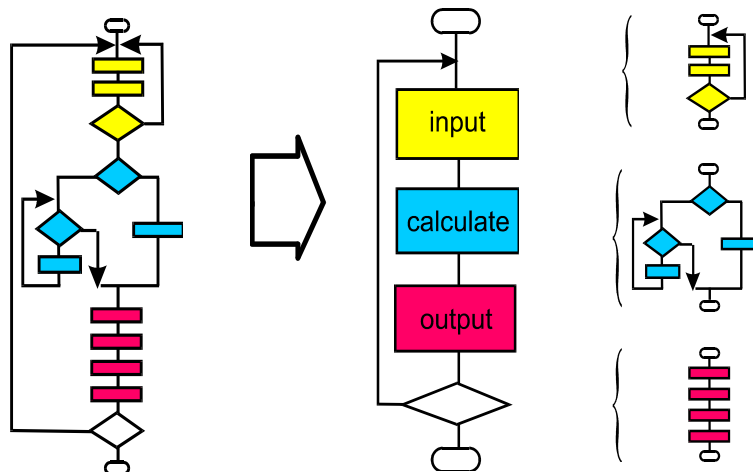


- Modular design may be *top-down*
- Subprograms implement modular designs

## Modularity and subprograms

- *Divide and conquer*: a problem solving strategy that breaks problems down into subproblems, each with a simpler solution
- In this scenario, *subprograms* (“functions” in JavaScript lingo) solve subproblems
- JavaScript functions, once *defined* (spelled out), may be *called* (invoked)
- Interfaces between subprogram and rest of program: *parameters*; *return values*

## Modular decomposition: case study

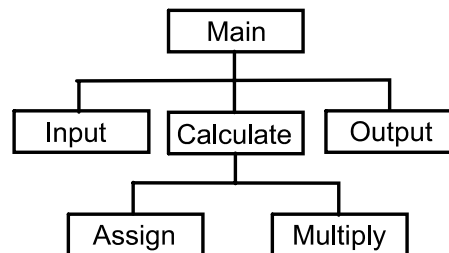


*Separate modules are easier to understand.*

## Module hierarchy diagrams

- *Example:*  
*Main invokes Input, Calculate, and Output;*  
*Calculate calls Assign and Multiply*

Main  
A. Input  
B. Calculate  
    1. Assign  
    2. Multiply  
C. Output



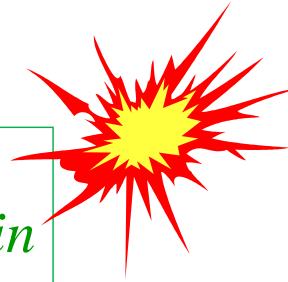
- A module hierarchy chart shows module dependencies, whereas a flowchart shows order of execution.

## 2. Algorithm design tools

- Essential properties of *algorithmic thinking* (L. Snyder):
  - Specification of input, output, and relationship between them
  - Definiteness (deterministic sequence of operations)
  - Effectiveness (doability)
  - Finiteness

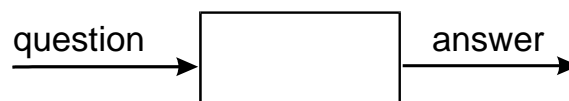
## Algorithm:

*A precise plan to convert input to output in a finite number of steps*



- Program designs use algorithms
- Much computation is algorithmic
- Flowcharts and pseudocode can express algorithms

## Expressing an algorithm or process



- Natural language
- Language of mathematics
- Design notations
  - Flowchart
  - Pseudocode
  - Unified Modeling Language (UML)
- Programming language (JavaScript, Java, C++...)
- Machine and assembler languages

## Two notations for low-level design

- Both notations show order of execution

- **Pseudocode**
  - informal
  - precise
  - text outline format

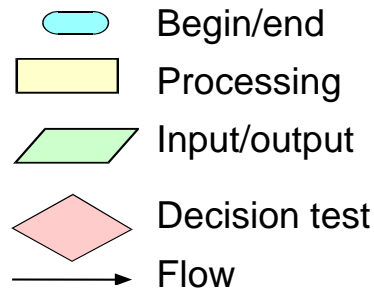
**Equivalence of values:**

$$a = b^2 + 5$$

**Assignment:**

$$y \leftarrow x_1 + x_2$$

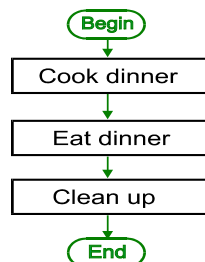
- **Flowcharts**
  - graphical
  - shapes denote steps
  - arrows show flow of control



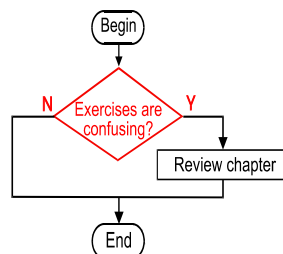
## Control structures

All algorithms may be built from three basic *control structures*:

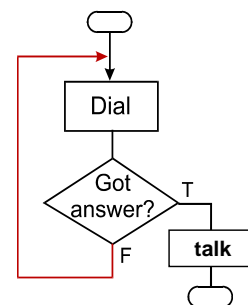
### Sequence



### Branch



### Loop



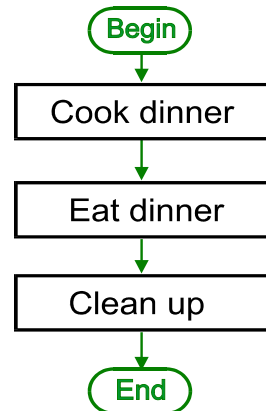
## The sequence control structure

*Problem:*  
Accomplish dinner routine

*Pseudocode:*

1. Cook dinner
2. Eat dinner
3. Clean up

*Flowchart:*



## An algorithm to add numbers

1. Prompt for integers  $input1$ ,  $input2$
2.  $sum \leftarrow input1 + input2$
3. Display  $sum$

- In pseudocode and flowcharts, the symbol  $\leftarrow$  stands for assignment of a value to a variable
- An algorithm is almost always *general purpose*: it works with *variables*, not only specific constant values

## The *decision* control structure

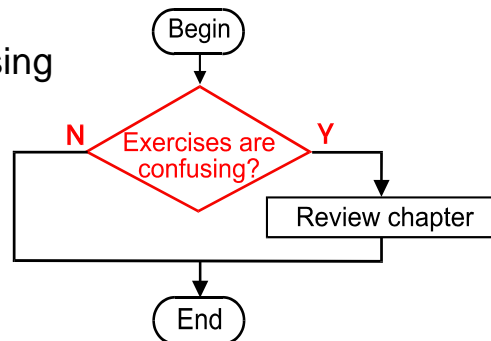
*Problem:*

Prepare to do homework

*Pseudocode:*

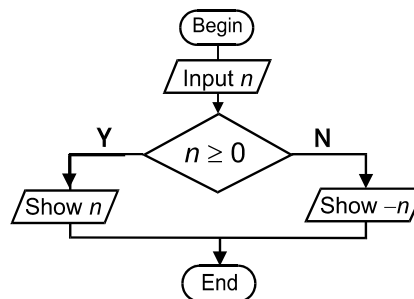
If exercises are confusing  
review chapter

*Flowchart:*



## Finding absolute value

**Input  $n$**   
**If  $n \geq 0$**   
**display  $n$**   
**otherwise**  
**display  $(-n)$**



- With the *branch* control structure, one and only one of the alternatives executes
- In pseudocode, the subordinate (conditional) steps are normally indented

## IF in spreadsheets

Excel has an IF function that yields a value conditional on a cell value

`= if(c4 > 0, "yes", "no")`

yields “yes” as cell value if cell c4 has a value greater than 0

## 3. The loop control structure

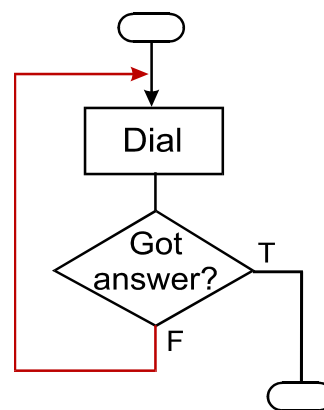
### *Problem:*

Telephone someone

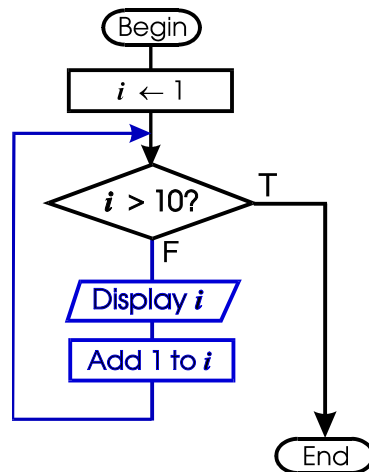
### *Pseudocode:*

Repeat  
Dial number  
until someone answers

### *Flowchart:*



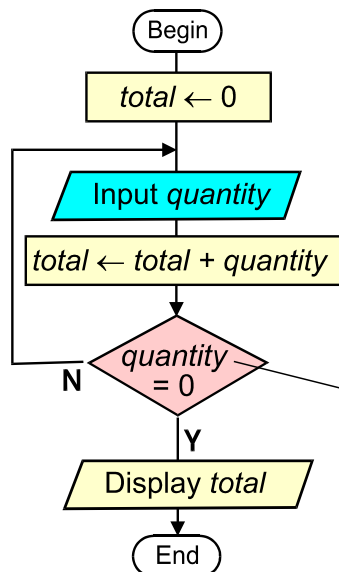
## Counting to 10



while  $i \leq 10$   
 display  $i$   
 $i \leftarrow i + 1$

- A *counted* loop; counter is  $i$

## A sentinel-controlled loop



*Not an algorithm,*  
 because input  
 alternates with  
 processing

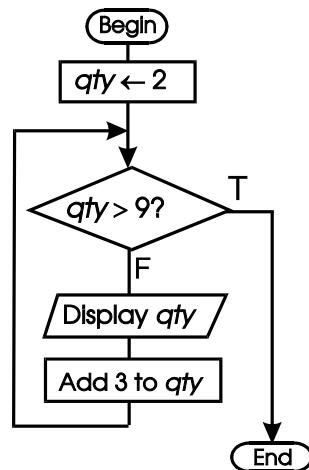
*sentinel value*

## Tracing an algorithm or process

- Allows designer to check result of algorithm, including internal (undisplayed) values
- Use one column per value traced; one row per loop iteration.
- *Example* (See previous slide), assuming input are 3, 2, 1, 0:

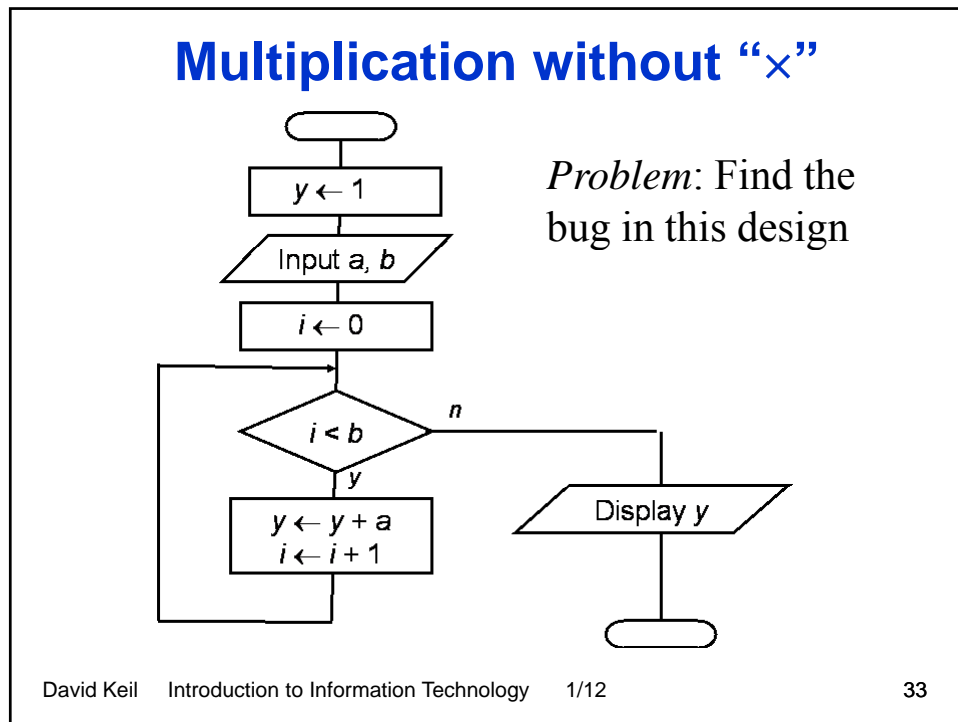
<i>quantity</i>	<i>total</i>	<i>output</i>
	0	
3	3	
2	5	
1	6	
0	6	6

## Algorithm without input



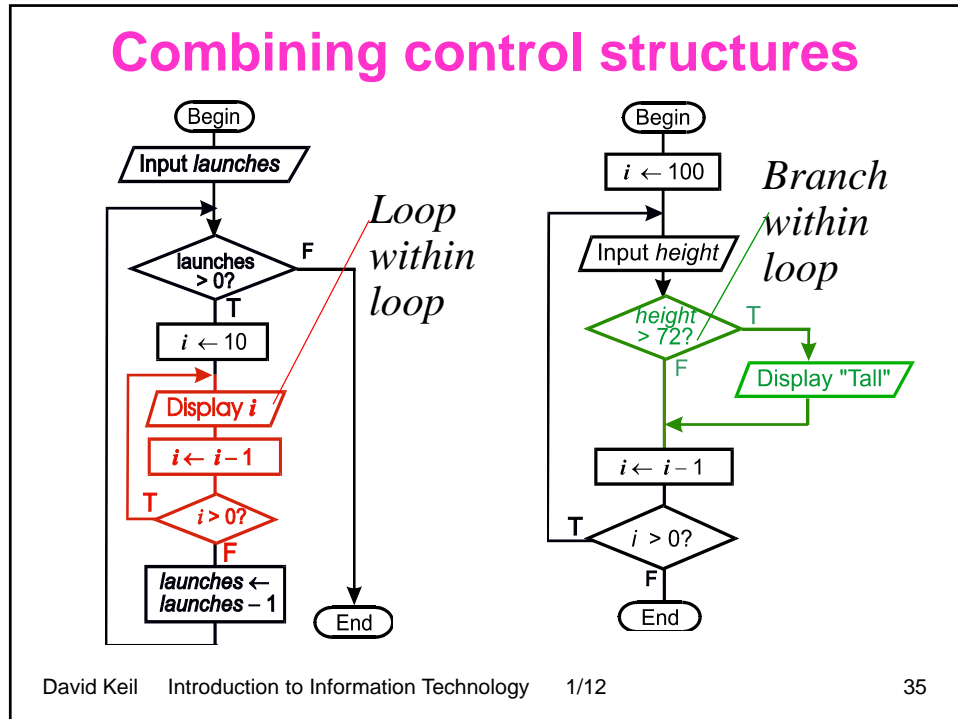
Trace:

<i>qty</i>	<i>output</i>
2	2
5	5
8	8
11	



## Why trace?

- Computer programs don't display all their internal workings
- To find and fix a car problem, the mechanic must look under the hood
- A trace displays the values of all variables as they change
- Tracing is crucial in debugging programs and systems



## Branch within loop

- *Problem:* Is a vote unanimous?
- *Solution:*

```

input num_votes, vote
is_unan ← true, prev ← vote
for i ← 1 to num_votes do
    if prev ≠ vote
        is_unan ← false
    prev ← vote
    input vote
if is_unan
    display "unanimous"
else display "not unanimous"

```

### Debugging in problem solving

- IT often displays *faults* where system fails to produce desired results
- *Debugging*: “figuring out why a process or system doesn’t work properly” (L. Snyder, 2006) – and fixing the error!
- Three causes of error:
  - Faulty input *data* (e.g., typo)
  - Faulty *command* (e.g., user misunderstands command syntax)
  - Faulty *system* (e.g., program bug)

### Debugging skills

Debugging data, commands, or system requires analytical and logical thinking and precision

- Is error reproducible?
- Localize the problem; e.g., to one device
- Avoid just “trying something”

## Debugging using trace

- *Problem:* Why does the pseudocode below fail to add the numbers from 1 to 5?

```
count ← 0
sum ← 0
while count < 5
  input x
  count ← count + 1
  sum ← sum + 1
display sum
```

- *Solution:* Trace the algorithm, see where an unexpected value occurs

## 4. JavaScript and event-driven software

- *JavaScript:* A procedural language
- The JavaScript text between the HTML tags `<script language = "Javascript">` and `</script>` will execute when browser displays HTML file
- *Motivation:*
  - Working with IT means thinking abstractly and concretely about data and operations
  - Design, coding, and testing of solutions are part of learning problem solving

## Generations of programming languages

- First generation: Machine languages (binary)
- 2<sup>nd</sup> generation: Assembler languages, processor specific
- 3<sup>rd</sup> generation: Procedural, high-level, hardware independent (C, BASIC, JavaScript)
- 4<sup>th</sup> generation: Nonprocedural query or report-generation languages (SQL, RPG)
- Declarative languages (Prolog)
- Object-oriented languages (C++, Java)
- Functional languages (Lisp)

## Problem specifications and user interfaces

- Designer must consider *assumptions* about
  - Problem domain (e.g., business, education, personal, healthcare)
  - User needs and expectations
- *Interface* refers to how application (e.g., at web site) appears and responds to user
- Most user interfaces today are *graphical*
- Implementation (coding) is partly independent of interface

## Event-driven design

- *Browsers* and most other apps are *interactive*, alternating input and output
- *Command-line environments*: URL line in browser, Google prompt, DOS or UNIX prompt
- *Features of graphical user interfaces*: windows, icons, menus, dialog boxes, buttons
- *Common feature*: User generates *events*, e.g., clicks, drags, keystrokes, timeouts
- Forms of interaction in browser: hyperlinks; embedding of event-based JavaScript programs in HTML files

## Simple JavaScript example

```
<html>  <!-- hello.htm -->
  <head><title>DK-Hello</title></head>
  <body>63.120 says Hello!
    <script language="JavaScript">
      alert("hello");
    </script>
  </body>
</html>
```

- Displays “hello” in an alert box (a kind of dialog)
- **alert** is a JavaScript function (a kind of procedure)
- JavaScript may be used after **script** tag

## Simple button

```
<html> <!--button.htm-->
<head><title>63120 Hello</title></head>
<body>
  <input type=button value = "Hello"
        onClick = 'alert("Hi")'>
</body></html>
```

- This code displays “Hi” when “Hello” button is pressed
- **<input>** tag defines an input button object
- *Event handler*: code that specifies application’s response to a particular event, such as user click on a button

## Counting button clicks

```
<html> <head><title>yes-no counter</title></head> <body>
/* count-yes.htm Displays Yes, No buttons for user to
click, counts # clicks on each. Four 'event-handlers'
specify the program's response to four different input
events: Yes, No, Stats, Reset. When user presses 'Yes'
button, variable 'num_yes' is incremented. */
<script language="JavaScript">
  var num_yes=0, num_no=0; // Variables
</script>
<td><input type=button value = "Yes"
          onClick = 'num_yes = num_yes + 1'></td>
<td><input type=button value = "No"
          onClick = 'num_no = num_no + 1'></td>
<td><input type=button value = "Stats"
          onClick = 'alert("Yes: "+num_yes + " No:
"+num_no)'"></td>
<td><input type=button value = "Reset"
          onClick = 'num_yes = num_no = 0'></td>
</body> </html>
```

## Text input/output

```
<head><title>Input echo</title></head> <body>
  <form name="Input"><table>
    <!-- Display prompt and get input:-->
    <td>Enter your user name:
    <!-- Generate input-box:-->
      <input type=text    name=user
          value=""    size = 15> </td>
    <!-- At button-press, display message:-->
    <td><input type=button value="Done"
      onClick = 'alert("Hello " +
user.value)'\> </td>
    <!-- Assigning a value to onClick defines
      JavaScript response to button click -->
  </table></form> </body>
```

## Statements

- *Statements* are *executable* (or are variable declarations)
- Statements include
  - *assignment* **a = 4;**
  - compound statements { in braces }
  - *if, if ... else*
  - *while*
- *Simple* statements, e.g., assignment, terminate with “.”

## JavaScript variables

- Variables have *name*, *type*, *value*
- Must *declare* a variable to use it  
`var a;`
- Valid *types* (interpretation of bits):
  - Number: `var a=2, b=3;`
  - String: `var name = "Bill";`
  - Truth value: `var greater = (a > b);`
- *Assignment* can change variable's value:  
`a = b * 4;`

## Expressions

- *Expressions* have *values* and are used in statements
- Expressions are *literals*, *variables*, or *function calls*, or may be formed with *arithmetic operators* (+, -, \*, /, %)
- *Boolean* expressions have truth values and may be built with *relational operators* (==, !=, <, >, <=, >=) or *logical operators* (!, &&, ||)

## Branches and loops in JavaScript

- *Branch:*

```
if (a > b)
    max = a;
else
    max = b;
```

See example code  
from D. Keil,  
C. Breuning

- *Loop:*

```
i = 1;
while(i < 10)
{
    sum = sum + i;
    i = i + 1;
}
```

## Testing and debugging

- Software and web sites require testing before deployment
- Testing is often done by quality assurance departments
- All software writing entails error and *debugging*
- JavaScript is easy to test on a browser, but the browser does not supply error locations or other diagnostics

## Example with branch control structure

```
<html> <head><title>Yes-no with memory</title></head>
<body>
  <script language="JavaScript">
    /* remember.htm  Displays 2 buttons for user to click,
       tells which was clicked most recently */
    var most_recent="A", message="Last you clicked was ";
  </script>
  <td><input type=button value = "A"
            onClick = 'most_recent="A"'></td>
  <td><input type=button value = "B"
            onClick = 'most_recent="B"'></td>
  <td><input type=button value = "Check" onClick =
        'if (most_recent == "A") alert(message+"A");
        else alert(message+"B")'> </td>
</body>
</html>
```

[remember.htm]

## Example with loop

```
<html> <! power.htm> <head><title>Input exponentiator</title></head>
<body> <script language = "JavaScript"> var i,y; </script>
  <form name="Input"><table>
    <! Display prompt and get input:>
    <td>Enter a base: <! Generate input-box:>
      <input type=text name=x1 value="0" size = 8> </td>

    <td>Enter an exponent:
      <! Generate input-box:>
      <input type=text name=x2
        value="0" size = 8> </td>

    <td><input type=button value="Done" onClick = '
      i = parseInt(x2.value);
      y = 1;
      while(i > 0) {
        y = y * parseInt(x1.value);
        i = i - 1;
      }
      alert(x1.value + "^" + x2.value +
        " = " + y);
    '> </td>
  </table></form> </body> </html>
```

### Concepts

algorithm	maintenance
analysis	modular design
branch	object-oriented design
class	pseudocode
coding	sequence
control structure	specification
debugging	system
decision	system development life cycle
design	testing
flowchart	trace
information system	Unified Modeling Language
loop	use case

### References

- A. Evans, K. Martin, M. A. Poatsy. *Go! Technology in Action*, 6<sup>th</sup> ed. Prentice Hall, 2010.
- J. Parsons and D. Oja. *Computer Concepts*, 9<sup>th</sup> ed. Thomson, 2007.
- L. Snyder. *Fluency with Information Technology*. Addison Wesley, 2006.
- G. Schneider and J. Winters. *Applying Use Cases: A Practical Guide*. Addison Wesley, 1998.