

Programming language components

- **syntax: grammar rules for defining legal statements**
 - what's grammatically legal? how are things built up from smaller things?
- **semantics: what things mean**
 - what do they compute?
- **statements: instructions that say what to do**
 - compute values, make decisions, repeat sequences of operations
- **variables: places to hold data in memory while program is running**
 - numbers, text, ...
- **most languages are higher-level and more expressive than the assembly language for the toy machine**
 - statements are much richer, more varied, more expressive
 - variables are much richer, more varied
 - grammar rules are more complicated
 - semantics are more complicated
- **but it's basically the same idea**

Javascript components

- **Javascript language**
 - statements that tell the computer what to do
get user input, display output, set values, do arithmetic,
test conditions, repeat groups of statements, ...
- **libraries, built-in functions**
 - pre-fabricated pieces that you don't have to create yourself
alert, prompt, math functions, text manipulation, ...
- **access to browser and web pages**
 - buttons, text areas, images, page contents, ...
- **you are not expected to remember syntax or other details**
- **you are not expected to write code in exams**
(though a bit in problem sets and labs)
- **you are expected to understand the ideas**
 - how programming and programs work
 - figure out what a tiny program does or why it's broken

Basic example 0: echo a name (name.html)

- Javascript code appears in HTML file between `<script>` tags
 `<script> ... </script>`
- this example shows variable, dialog box

```
<html>
<body>
<P> nam2.html: echoes a name
<script>
    var name;
    name = prompt("What's your name?");
    alert("hello, " + name);
</script>
```

Basic example 1: join 2 names (name2.html)

- Javascript code appears in HTML file between `<script>` tags
 `<script> ... </script>`
- shows variables, dialog boxes, an operator

```
<html>
```

```
<body>
```

```
<P> name2.html: joins 2 names
```

```
<script>
```

```
    var firstname, secondname, result;
```

```
    firstname = prompt("Enter first name");
```

```
    secondname = prompt("Enter last name");
```

```
    result = firstname + secondname; // + means "join" here
```

```
    alert("hello, " + result); // and here
```

```
</script>
```

Basic example 2: add 2 numbers (add2.html)

- dialog boxes, variables, arithmetic, conversion

```
<html>
<body>
<P> add2.html: adds 2 numbers
<script>
    var num1, num2, sum;
    num1 = prompt("Enter first number");
    num2 = prompt("Enter second number");
    sum = parseInt(num1) + parseInt(num2); // "+" means "add"
    alert(sum);
</script>
```

`parseInt(...)` converts a sequence of characters into its integer value
there's also a `parseFloat(...)` for floating point numbers

Adding up lots of numbers: addup.html

- variables, operators, expressions, assignment statements
- while loop, relational operator (`!=` "not equal to")

```
<html>
<body>
<script>
    var sum = 0;
    var num;
    num = prompt("Enter new value, or 0 to end");
    while (num != 0) {
        sum = sum + parseInt(num);
        num = prompt("Enter new value, or 0 to end");
    }
    alert("Sum = " + sum);
</script>
```

Find the largest number: max.html

- needs an If to test whether new number is bigger
- needs another relational operator
- needs parseInt or parseFloat to treat input as a number

```
var max = 0;
var num;
num = prompt("Enter new value, or 0 to end");
while (num != 0) {
    if (parseFloat(num) > max)
        max = num;
    num = prompt("Enter new value, or 0 to end");
}
document.write("<P> Max = " + max);
```

Variables, constants, expressions, operators

- a *variable* is a place in memory that holds a value
 - has a **name** that the programmer gave it, like **sum** or **Area** or **n**
 - in Javascript, can hold any of multiple types, most often numbers like **1** or **3.14**, or sequences of characters like **"Hello"** or **"Enter new value"**
 - always has a **value**
 - has to be set to some value initially before it can be used
 - its value will generally change as the program runs
 - ultimately corresponds to a location in memory
 - but it's easier to think of it just as a name for information
- a *constant* is an unchanging literal value like **3** or **"hello"**
- an *expression* uses operators, variables and constants to compute a value
 - $3.14 * \text{rad} * \text{rad}$
- *operators* include **+ - * /**

Computing area: area.html

```
var rad, area;
rad = prompt("Enter radius");
while (rad != null) {
    area = 3.14 * rad * rad;
    document.write("<P> radius = " + rad + ", area = " + area);
    rad = prompt("Enter radius");
}
```

- **how to terminate the loop**
 - 0 is a valid data value
 - `prompt()` returns null for Cancel and "" for OK without typing any text
- **string concatenation to build up output line**
- **there is no exponentiation operator so we use multiplication**

Types, declarations, conversions

- variables have to be declared in a var statement
- each variable holds information of a specific type
 - really means that bits are to be interpreted as info of that type
 - internally, 3 and 3.00 and "3.00" are represented differently
- Javascript usually infers types from context, does conversions automatically
 - "Sum = " + sum
- sometimes we have to be explicit:
 - `parseInt(...)` if can't tell from context that string is meant as an integer
 - `parseFloat(...)` if it could have a fractional part

Making decisions and repeating statements

- **if-else statement makes decisions**
 - the Javascript version of decisions written with ifzero, ifpos, ...

```
if (condition is true) {  
    do this group of statements  
} else {  
    do this group of statements instead  
}
```

- **while statement repeats groups of statements**
 - a Javascript version of loops written with ifzero and goto

```
while (condition is true) {  
    do this group of statements  
}
```

if-else examples (sign.html)

- can include else-if sections for a series of decisions:

```
var num = prompt("Enter number");
while (num != null) {
    num = parseInt(num);
    if (num > 0) {
        alert(num + " is positive");
    } else if (num < 0) {
        alert(num + " is negative");
    } else {
        alert(num + " is zero");
    }
    num = prompt("Enter number");
}
```

"while loop" examples

- counting or "indexed" loop:

```
i = 1;
while (i <= 10) {
    // do something (maybe using the current value of i)
    i = i + 1;
}
```

- "nested" loops (while.html):

```
var n = prompt("Enter number");
while (n != null) {    // "!=" means "is not equal to"
    i = 0;
    while (i <= n) {
        document.write("<br>" + i + " " + i*i);
        i = i + 1;
    }
    n = prompt("Enter number");
}
```

Functions

- **a function is a group of statements that does some computation**
 - the statements are collected into one place and given a name
 - other parts of the program can "call" the function
 - that is, use it as a part of whatever they are doing
 - can give it values to use in its computation (arguments or parameters)
 - computes a value that can be used in expressions
 - the value need not be used
- **Javascript provides some useful built-in functions**
 - e.g., prompt, alert, ...
- **you can write your own functions**

Function examples

- **syntax**

```
function name (list of "arguments" ) {  
    the statements of the function  
}
```

- **function definition:**

```
function area(r) {  
    return 3.14 * r * r;  
}
```

- **using ("calling") the function:**

```
rad = prompt("Enter radius");  
alert("radius = " + rad + ", area = " + area(rad));  
  
alert("area of CD =" + area(2.3) - area(0.8));
```

Ring.html

```
var r1, r2;
r1 = prompt("Enter radius 1");
while (r1 != null) {
    r2 = prompt("Enter radius 2");
    alert("area = " + (area(r1) - area(r2))); // parens needed!
    r1 = prompt("Enter radius 1");
}

function area(r) {
    return 3.14 * r * r;
}
```


Why use functions?

- **if a computation appears several times in one program**
 - a function collects it into one place
- **breaks a big job into smaller, manageable pieces**
 - that are separate from each other
- **defines an interface**
 - implementation details can be changed as long as it still does the same job
 - different implementations can interoperate
- **multiple people can work on the program**
- **a way to use code written by others long ago and far away**
 - most of Javascript's library of useful stuff is accessed through functions
- **a good library encourages use of the language**

Summary: elements of (most) programming languages

- constants: literal values like 1, 3.14, "Error!"
- variables: places to store data and results during computing
- declarations: specify name (and type) of variables, etc.
- expressions: operations on variables and constants to produce new values
- assignment: store a new value in a variable
- statements: assignment, input/output, loop, conditional, call
- conditionals: compare and branch; if-else
- loops: repeat statements while a condition is true
- functions: package a group of statements so they can be called/used from other places in a program
- libraries: functions already written for you

How Javascript works

- recall the process for Fortran, C, etc.:
 compiler -> assembler -> machine instructions
- **Javascript is analogous, but differs significantly in details**
- **when the browser sees Javascript in a web page (<script> tags)**
 - passes the Javascript program to a Javascript compiler
- **Javascript compiler**
 - checks for errors
 - compiles the program into instructions for something like the toy machine, but richer, more complicated, higher level
 - runs a simulator program (like the toy) that interprets these instructions
- **simulator is often called an "interpreter" or a "virtual machine"**
 - probably written in C or C++ but could be written in anything
- **browser and simulator interact**
 - when an event like click happens, browser tells Javascript ("onClick")
 - Javascript tells browser to do things (e.g., pop up dialog box for `alert`)

The process of programming

- what we saw with Javascript or Toy is like reality, but very small
- **figure out what to do**
 - start with a broad specification
 - break into smaller pieces that will work together
 - spell out precise computational steps in a programming language
- **build on a foundation (rarely start from scratch)**
 - a programming language that's suitable for expressing the steps
 - components that others have written for you
 - functions from libraries, major components, ...
 - which in turn rest on others, often for several layers
 - runs on software (the operating system) that manages the machine
- **it rarely works the first time**
 - test to be sure it works, debug if it doesn't
 - evolve as get a better idea of what to do, or as requirements change

Real-world programming

- **the same thing, but on a grand scale**
 - programs may be millions of lines of code
 - typical productivity: 1-10K lines/year/programmer
 - thousands of people working on them
 - lifetimes measured in years or even decades
- **big programs need teams, management, coordination, meetings, ...**
- **schedules and deadlines**
- **constraints on how fast the program must run, how much memory it can use**
- **external criteria for reliability, safety, security, interoperability with other systems, ...**
- **maintenance of old ("legacy") programs is hard**
 - programs must evolve to meet changing environments and requirements
 - machines and tools and languages become obsolete
 - expertise disappears