JavaScript Basics

Jerry Cain
CS 106AX
September 25, 2024
slides leveraged from those written by Eric Roberts

Control Statements

Statement Types in JavaScript

- Statements in JavaScript fall into three basic types:
  - Simple statements
  - Compound statements
  - Control statements
- ***Simple statements*** are typically assignments, function calls, or applications of the `++` or `--` operators. Simple statements are always terminated with a semicolon.
- ***Compound statements*** (also called ***blocks***) are sequences of statements enclosed in curly braces.
- ***Control statements*** fall into two categories:
  - ***Conditional statements*** that require some test be evaluated
  - ***Iterative statements*** that specify repetition

Boolean Expressions

- JavaScript defines two types of operators that work with Boolean data: ***relational operators*** and ***logical operators***.
- There are six relational operators that compare values of other types and produce a **`true`**/**`false`** result:

`===`	Equals	**`!==`**	Not equals
`<`	Less than	**`<=`**	Less than or equal to
`>`	Greater than	**`>=`**	Greater than or equal to

For example, the expression **`n <= 10`** has the value **`true`** if **`n`** is less than or equal to 10 and the value **`false`** otherwise.

- There are also three logical operators:

`&&`	Logical AND	**`p && q`** means both **`p`** and **`q`**
`||`	Logical OR	**`p || q`** means either **`p`** or **`q`** (or both)
`!`	Logical NOT	**`!p`** means the opposite of **`p`**

Notes on the Boolean Operators

- Remember that JavaScript uses `=` for assignment. To test whether two values are equal, you must use the `===` operator.
- It is not legal in JavaScript to use more than one relational operator in a single comparison. To express the idea embodied in the mathematical expression

  $$0 \leq x \leq 9$$

  you need to make both comparisons explicit, as in

  ```
  0 <= x && x <= 9
  ```

- The `||` operator means *either or both,* which is not always clear in the English interpretation of *or.*
- Be careful when you combine the `!` operator with `&&` and `||` because the interpretation often differs from informal English.

Short-Circuit Evaluation

- JavaScript evaluates the **`&&`** and **`||`** operators using a strategy called ***short-circuit mode*** in which it evaluates the right operand only if it needs to do so.
- For example, if **`n`** is 0, the right operand of **`&&`** in

  ```
  n !== 0 && x % n === 0
  ```

  is not evaluated at all because **`n !== 0`** is **`false`**. Because the expression

  `false && ` *anything*

  is always **`false`**, the rest of the expression no longer matters.
- One of the advantages of short-circuit evaluation is that you can use **`&&`** and **`||`** to prevent errors. If **`n`** were 0 in the earlier example, evaluating **`x % n`** would result in a division by zero.

The `if` Statement

- The simplest of the control statements is the **`if`** statement, which occurs in two forms. You use the first when you need to perform an operation only if a particular condition is true:

```
if (condition) {
   statements to be executed if the condition is true
}
```

- You use the second form whenever you need to choose between two alternative paths, depending on whether the condition is true or false:

```
if (condition) {
   statements to be executed if the condition is true
} else {
   statements to be executed if the condition is false
}
```

Functions Involving Control Statements

- The body of a function can contain statements of any type, including control statements. As an example, the following function uses an **`if`** statement to find the larger of two values:

```
function max(x, y) {
   if (x > y) {
      return x;
   } else {
      return y;
   }
}
```

- As this example makes clear, **`return`** statements can be used at any point in the function and may appear more than once.

The `switch` Statement

The **`switch`** statement provides a convenient syntax for choosing among a set of possible paths:

```
switch ( expression ) {
   case v1:
      statements to be executed if expression is equal to v1
      break;
   case v2:
      statements to be executed if expression is equal to v2
      break;
   . . . more case clauses if needed . . .
   default:
      statements to be executed if no values match
      break;
}
```

Example of the `switch` Statement

The **`switch`** statement is useful when a function must choose among several cases, as in the following example:

```
function monthName(month) {
   switch (month) {
    case  1: return "January";
    case  2: return "February";
    case  3: return "March";
    case  4: return "April";
    case  5: return "May";
    case  6: return "June";
    case  7: return "July";
    case  8: return "August";
    case  9: return "September";
    case 10: return "October";
    case 11: return "November";
    case 12: return "December";
    default: return undefined;
   }
}
```

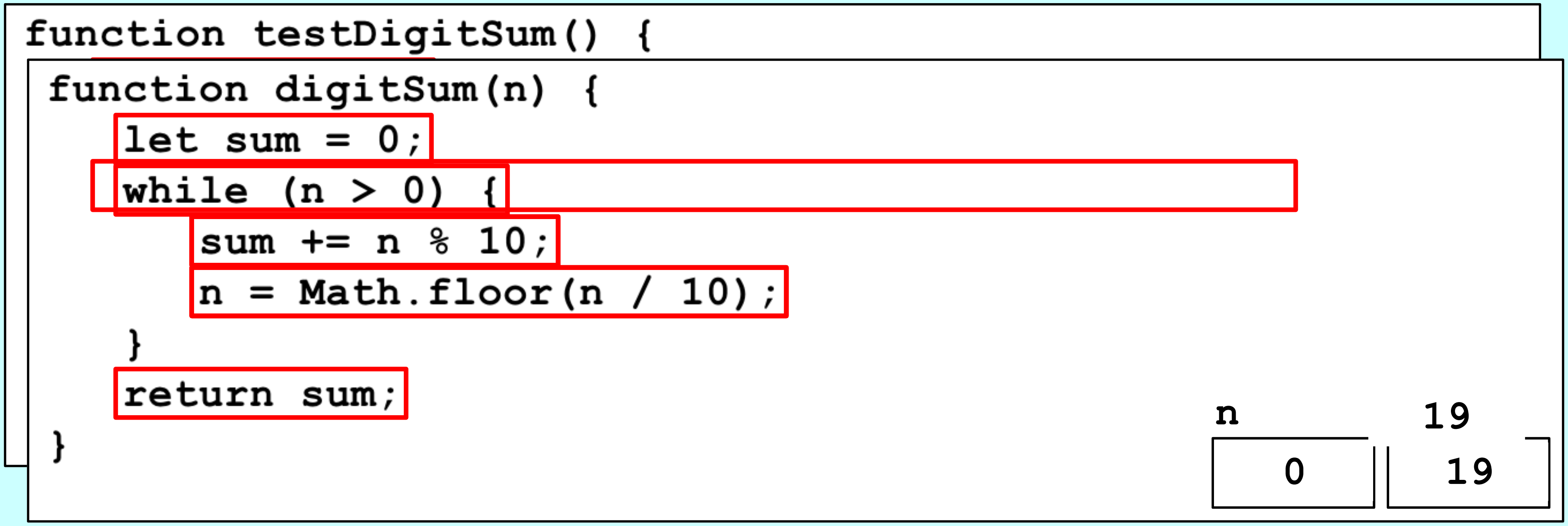
The `while` Statement

- The **`while`** statement is the simplest of JavaScript's iterative control statements and has the following form:

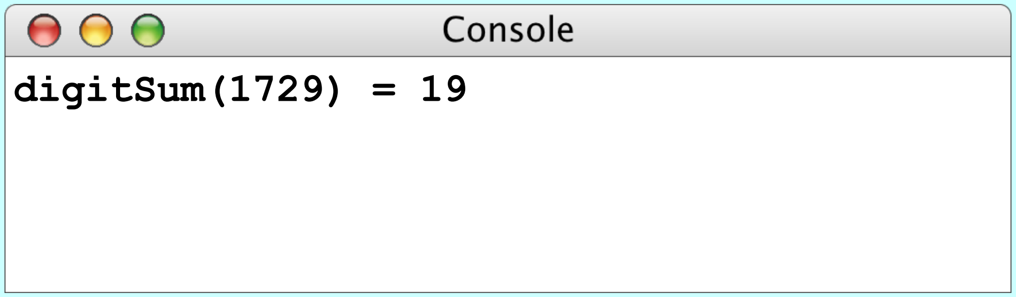
```
while ( condition ) {
        statements to be repeated
}
```

- When JavaScript encounters a **`while`** statement, it begins by evaluating the condition in parentheses.
- If the value of *condition* is **`true`**, JavaScript executes the statements in the body of the loop.
- At the end of each cycle, JavaScript reevaluates *condition* to see whether its value has changed. If *condition* evaluates to **`false`**, JavaScript exits from the loop and continues with the statement following the end of the **`while`** body.

The **digitSum** Function



```
function testDigitSum() {
function digitSum(n) {
    let sum = 0;
    while (n > 0) {
        sum += n % 10;
        n = Math.floor(n / 10);
    }
    return sum;
}
```



```
digitSum(1729) = 19
```

The `for` Statement

- The `for` statement in JavaScript is a powerful tool for specifying the structure of a loop independently from the operations the loop performs. The syntax looks like this:

```
for ( init ; test ; step ) {
      statements to be repeated
}
```

- JavaScript evaluates a `for` statement as follows:

1. Evaluate *init*, which typically declares a ***control variable***.
2. Evaluate *test* and exit from the loop if the value is `false`.
3. Execute the statements in the body of the loop.
4. Evaluate *step,* which usually updates the control variable.
5. Return to step 2 to begin the next loop cycle.

Exercise: Reading `for` Statements

Describe the effect of each of the following **`for`** statements:

1.
```
for (let i = 1; i <= 10; i++)
```

This statement executes the loop body ten times, with the control variable **`i`** *taking on each successive value between 1 and 10.*

2.
```
for (let i = 0; i < n; i++)
```

This statement executes the loop body **`n`** *times, with* **`i`** *counting from* `0` *to* **`n-1`**. *This version is the standard Repeat-n-Times idiom.*

3.
```
for (let n = 99; n >= 1; n -= 2)
```

This statement counts backward from 99 to 1 by twos.

4.
```
for (let x = 1; x <= 1024; x *= 2)
```

This statement executes the loop body with the variable **`x`** *taking on successive powers of two from 1 up to 1024.*

The `factorial` Function

- The ***factorial*** of a number *n* (which is usually written as *n*! in mathematics) is defined to be the product of the integers from 1 up to *n*. Thus, 5! is equal to 120, which is $1 \times 2 \times 3 \times 4 \times 5$.
- The following function definition uses a **`for`** loop to compute the factorial function:

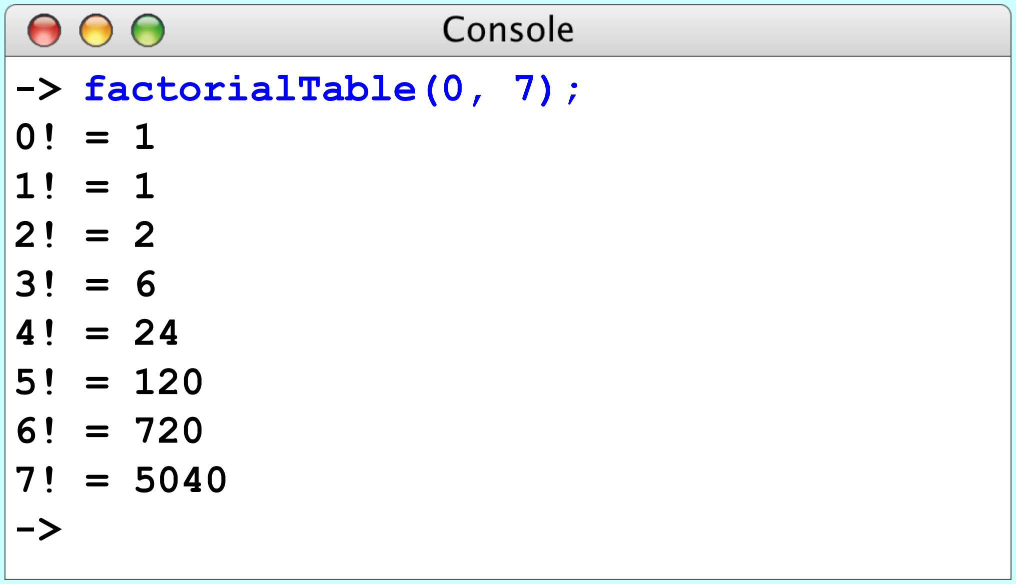
```
function fact(n) {
   let result = 1;
   for (let i = 1; i <= n; i++) {
      result = result * i;
   }
   return result;
}
```

The **factorialTable** Function

```
function factorialTable(min, max) {
  function fact(n) {
     let result = 1;
     for (let i = 1; i <= n; i++) {
        result = result * i;
     }
     return result;
  }
```

5040

n	result	8
7	5040	8



```
-> factorialTable(0, 7);
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
->
```

Comparing `for` and `while`

- The `for` statement

```
for ( init ; test ; step ) {
    statements to be repeated
}
```

is functionally equivalent to the following code using `while`:

```
init;
while ( test ) {
    statements to be repeated
    step;
}
```

- The advantage of the `for` statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

Functions Mechanics

A Quick Review of Functions

- The concept of a function should be familiar to you from prior programming experience. All modern programming languages allow functions to be defined.
- At the most basic level, a ***function*** is a sequence of statements that has been collected together and given a name. The name makes it possible to execute the statements much more easily; instead of copying out the entire list of statements, you can just provide the function name.
- The following terms are useful when working with functions:
  - Invoking a function by name is known as ***calling*** that function.
  - The caller passes information to a function using ***arguments***.
  - When a function completes its operation, it ***returns*** to its caller.
  - A function gives information to the caller by ***returning a result***.

Review: Syntax of Functions

- The general form of a function definition is

```
function name(parameter list) {
    statements in the function body
}
```

  where *name* is the name of the function, and *parameter list* is a list of variables used to hold the values of each argument.

- You can return a value from a function by including one or more **`return`** statements, which are usually written as

```
return expression;
```

  where *expression* is an expression that specifies the value you want to return.

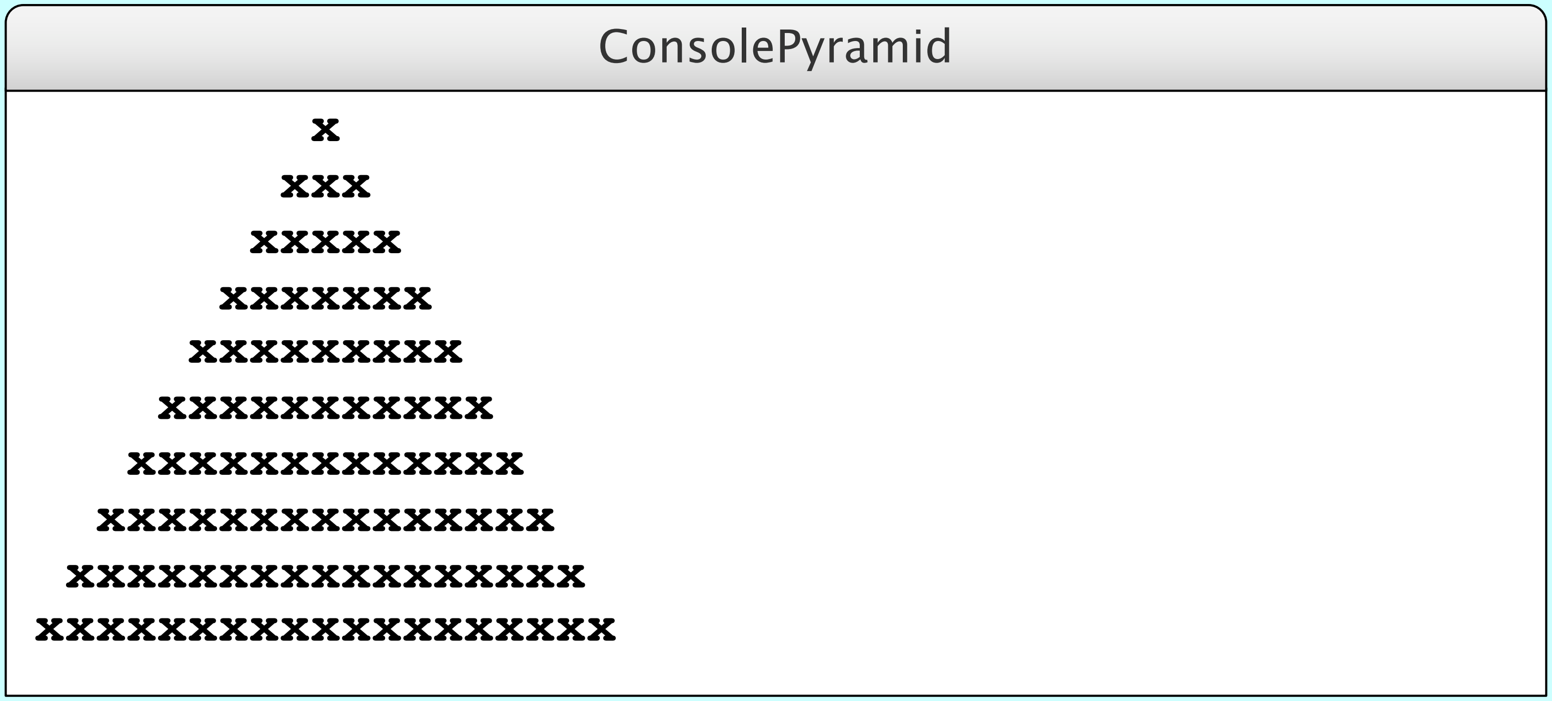
Nonnumeric Functions

- Although functions return a single value, that value can be of any type.
- Even without learning the full range of string operations covered in Chapter 7, you can already write string functions that depend only on concatenation, such as the following function that concatenates together **`n`** copies of the string **`str`**:

```
function concatNCopies(n, str) {
   let result = "";
   for (let i = 0; i < n; i++) {
      result += str;
   }
   return result;
}
```

Exercise: Console Pyramid

- Write a program that uses the **`concatNCopies`** function to display a pyramid on the console in which the bricks are represented by the letter **`x`**. The number of levels in the pyramid should be defined as the constant **`N_LEVELS`**.
- For example, if **`N_LEVELS`** is 10, the console output should look like this:



ConsolePyramid

```
         x
        xxx
       xxxxx
      xxxxxxx
     xxxxxxxxx
    xxxxxxxxxxx
   xxxxxxxxxxxxx
  xxxxxxxxxxxxxxx
 xxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxx
```

Predicate Functions

- Functions that return Boolean values play a central role in programming and are called ***predicate functions***. As an example, the following function returns **`true`** if the first argument is divisible by the second, and **`false`** otherwise:

```
function isDivisibleBy(x, y) {
   return x % y === 0;
}
```

- Once you have defined a predicate function, you can use it any conditional expression. For example, you can print the integers between 1 and 100 that are divisible by 7 as follows:

```
for (let i = 1; i <= 100; i++) {
   if (isDivisibleBy(i, 7)) {
      println(i);
   }
}
```

Using Predicate Functions Effectively

- New programmers often seem uncomfortable with Boolean values and end up writing ungainly code. For example, a beginner might write **`isDivisibleBy`** like this:

```
function isDivisibleBy(x, y) {
   if (x % y === 0) {
      return true;
   } else {
      return false;
   }
}
```

  While this code is technically correct, it is inelegant and should be replaced by **`return x % y === 0`**.

- A similar problem occurs when novices explicitly check to see whether a predicate function returns **`true`**. You should be careful to avoid such redundant tests in your own programs.

The Purpose of Parameters

"All right, Mr. Wiseguy," she said, "you're so clever, you tell us what color it should be."

—Douglas Adams, *The Restaurant at the End of the Universe, 1980*

- In general, functions perform some service to their callers. In order to do so, the function needs to know any details required to carry out the requested task.
- Imagine that you were working as a low-level animator at Disney Studies in the days before computerized animation and that one of the senior designers asked you to draw a filled circle. What would you need to know?
- At a minimum, you would need to know where the circle should be placed in the frame, how big to make it, and what color it should be. Those values are precisely the information conveyed in the parameters.

Libraries

- To make programming easier, all modern languages include collections of predefined functions. Those collections are called ***libraries***.
- For programming that involves mathematical calculations, the most useful library is the **`Math`** library, which includes a number of functions that will be familiar from high-school mathematics (along with many that probably aren't). A list of the most important functions appears on the next slide.
- In JavaScript, each of the functions in the **`Math`** library begins with the library name followed by a dot and then the name of the function. For example, the function that calculates square roots is named **`Math.sqrt`**.
- You call library functions just like any other function, so that calling **`Math.sqrt(16)`** returns the value 4.

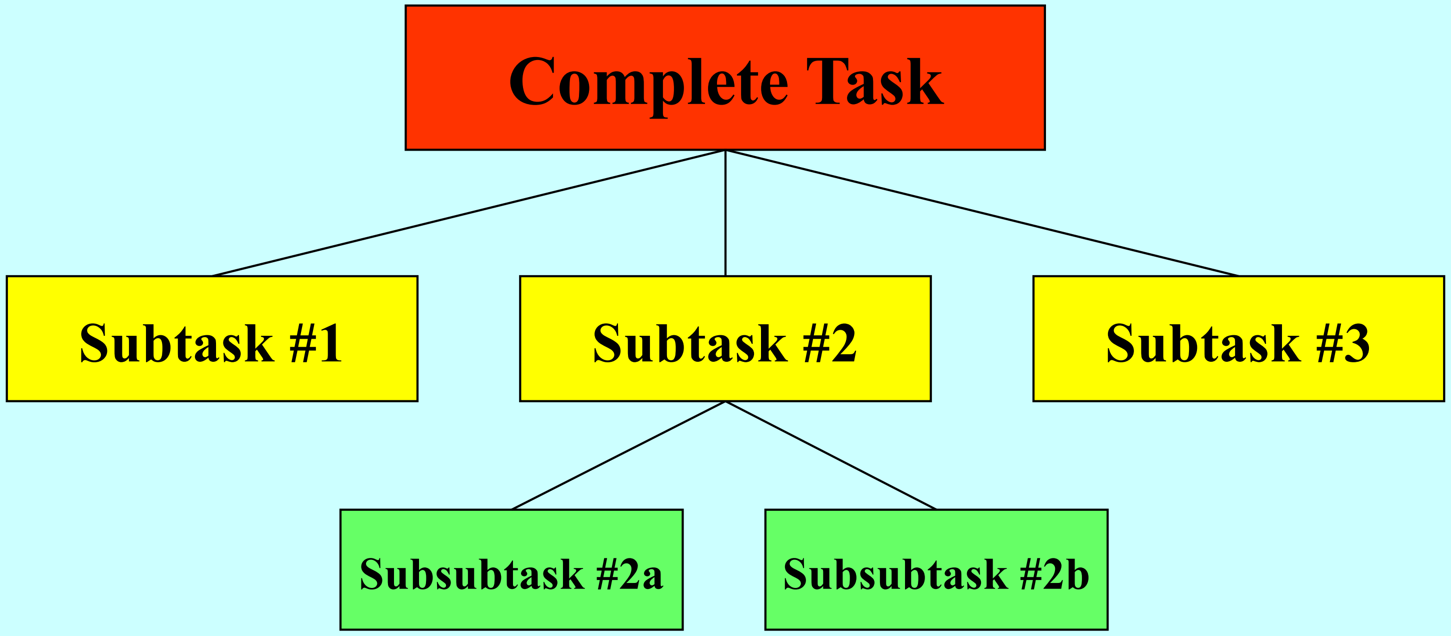
Useful Functions in the `Math` Library

`Math.PI`	The mathematical constant π
`Math.E`	The mathematical constant e
`Math.abs(`x`)`	The absolute value of x
`Math.max(`x`, `y`, . . .)`	The largest of the arguments
`Math.min(`x`, `y`, . . .)`	The smallest of the arguments
`Math.round(`x`)`	The closest integer to x
`Math.floor(`x`)`	The largest integer not exceeding x
`Math.log(`x`)`	The natural logarithm of x
`Math.exp(`x`)`	The inverse logarithm (e^x)
`Math.pow(`x`, `y`)`	The value x raised to the y power (x^y)
`Math.sin(`θ`)`	The sine of θ, measured in radians
`Math.cos(`θ`)`	The cosine of θ, measured in radians
`Math.sqrt(`x`)`	The square root of x
`Math.random()`	A random value between 0 and 1

Decomposition

Decomposition

- The most effective way to solve a complex problem is to break it down into successively simpler subproblems.
- You start by breaking the whole task down into simpler parts.
- Some of those tasks may themselves need subdivision.
- This process is called ***stepwise refinement*** or ***decomposition***.



Criteria for Choosing a Decomposition

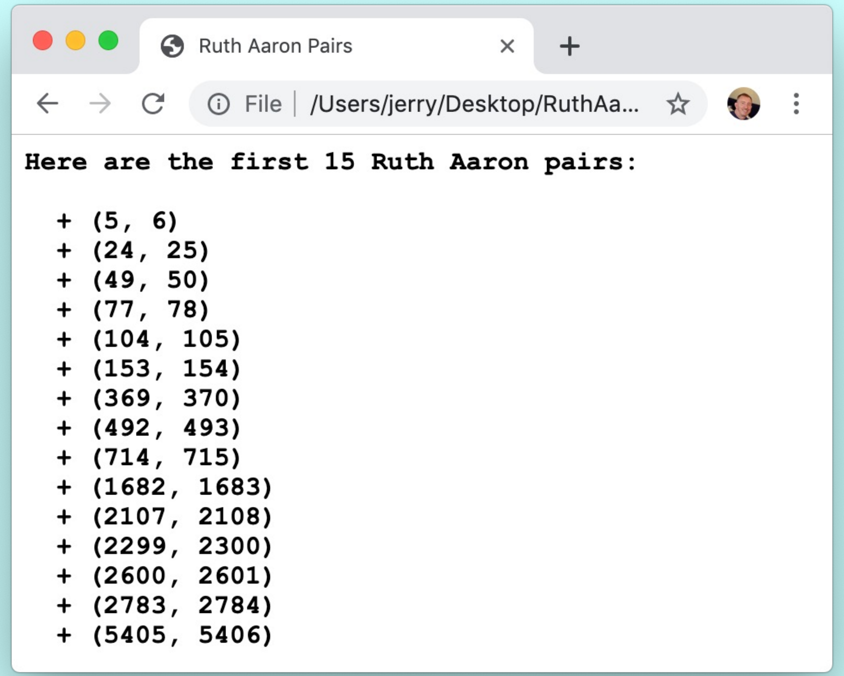
1. ***The proposed steps should be easy to explain.*** One indication that you have succeeded is being able to find simple names.

2. ***The steps should be as general as possible.*** Programming tools get reused all the time. If your methods perform general tasks, they are much easier to reuse.

3. ***The steps should make sense at the level of abstraction at which they are used.*** If you have a method that does the right job but whose name doesn't make sense in the context of the problem, it is probably worth defining a new method that calls the old one.

Exercise: Ruth-Aaron Pairs

- Write a program that lists the first **`NUM_PAIRS`** Ruth-Aaron pairs. In recreational mathematics, a Ruth-Aaron pair consists of two neighboring integers—e.g. 714 and 715—for which the sum of the *distinct* prime factors of each are equal.
- The pairs are named Ruth-Aaron as a nod to baseball greats Babe Ruth and Hank Aaron. Ruth held the record for most career home runs at 714 until Aaron's hit his 715th on April 8th, 1974.
- If **`NUM_PAIRS`** equals 15, the program should publish the output presented on the right.



```
Here are the first 15 Ruth Aaron pairs:

 + (5, 6)
 + (24, 25)
 + (49, 50)
 + (77, 78)
 + (104, 105)
 + (153, 154)
 + (369, 370)
 + (492, 493)
 + (714, 715)
 + (1682, 1683)
 + (2107, 2108)
 + (2299, 2300)
 + (2600, 2601)
 + (2783, 2784)
 + (5405, 5406)
```

The End