

Programming Language Samples

- For comparative purposes, the following fragments illustrate two algorithms written in 6 languages apiece: C, C++, Java, JavaScript, ML, and Perl
- The algorithms are:
 - ◇ *gcd* — Finds the greatest common divisor/denominator of two integers
 - ◇ *toRoman* — Converts an integer into its Roman numeral representation

GCD in C and C++

```
/*
 * This is the gcd() algorithm implemented in C.
 * The code actually looks identical in C++ as well,
 * since C++ “evolved” from C and gcd() doesn’t use
 * anything that is new to C++.
 */
int gcd(int i, int j) {
    while (i != j) {
        if (i > j)
            i -= j;
        else
            j -= i;
    }
    return i;
}
```

GCD in Java

```
package gcd;

/**
 * GCD is a simple Java implementation of a greatest common
 * denominator algorithm. Note how it has to live inside a
 * "class" construct.
 */
public class GCD {
    /**
     * Calculates the greatest common denominator of two integers.
     *
     * @param i the first integer
     * @param j the second integer
     * @return the GCD
     */
    public static int gcd(int i, int j) {
        // The actual algorithm.
        while (i != j) {
            if (i > j)
                i -= j;
            else
                j -= i;
        }

        return i;
    }

    /**
     * Console-driven driver program: note how this part isn't
     * the algorithm itself, but actually a separate unit that
     * uses the algorithm to calculate the gcd() of two numbers
     * provided by the user.
     */
    public static void main(String[] argv) {
        // Check for arguments.
        if (argv.length >= 2) {
            // Convert the arguments into integers.
            int i = Integer.parseInt(argv[0]);
            int j = Integer.parseInt(argv[1]);
            System.out.println("The GCD of " + argv[0] + " and " +
                argv[1] + " is " + gcd(i, j) + ".");
        } else {
            System.out.println("Please include two integers on the command line.");
        }
    }
}
```

GCD in JavaScript (with a Sample XHTML File Below)

```
/*
 * This is gcd() in JavaScript; it probably looks fairly
 * familiar to you at this point. The XHTML page below
 * assumes that this function is in a file called gcd.js.
 */
function gcd(i, j) {
    while (i != j) {
        if (i > j)
            i -= j;
        else
            j -= i;
    }

    return i;
}

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=iso-8859-1"/>
    <title>GCD Finder</title>

    <script type="text/javascript" src="gcd.js"></script>
    <script type="text/javascript">
function findGCD() {
    var output = document.getElementById("output");
    var form = document.getElementById("gcdInput");
    var i = parseInt(form.i.value);
    var j = parseInt(form.j.value);
    if (isNaN(i) || isNaN(j))
        output.innerHTML = "Either " + i + " or " + j + " (or both) is not an integer.";
    else
        output.innerHTML = gcd(i, j);
}
    </script>
</head>
<body>
    <form id="gcdInput" action="gcdInteractive.html">
        <p>Enter two integers:
            <input type="text" name="i"/>
            <input type="text" name="j"/>
            <input type="button" value="Find GCD" onclick="findGCD();"/>
        </p>
    </form>

    <p id="output"></p>
</body>
</html>
```

GCD in ML

```
(* The GCD algorithm, in a single recursive
   ML function. *)
fun gcd i j =
  if (i = j) then
    i
  else
    (if (i > j) then
      gcd (i - j) j
    else
      gcd i (j - i));
```

GCD in Perl

```
#
# This is a Perl implementation of a greatest common denominator algorithm.
#
use strict;
use warnings;

sub gcd {
  # Snag the arguments.
  my ($i, $j) = @_;

  # The actual algorithm.
  while ($i != $j) {
    if ($i > $j) {
      $i = $i - $j;
    } else {
      $j = $j - $i;
    }
  }

  return $i;
}
```

toRoman in C

```
/*
 * This is the toRoman() function written in C.
 */
#include <stdlib.h>
#include <string.h>

int toRoman(int n, char *numeral) {
    if (n <= 0) { return 1; }
    while (n >= 1000) { strcat(numeral, "M"); n -= 1000; }
    if (n >= 900) { strcat(numeral, "CM"); n -= 900; }
    if (n >= 500) { strcat(numeral, "D"); n -= 500; }
    if (n >= 400) { strcat(numeral, "CD"); n -= 400; }
    while (n >= 100) { strcat(numeral, "C"); n -= 100; }
    if (n >= 90) { strcat(numeral, "XC"); n -= 90; }
    if (n >= 50) { strcat(numeral, "L"); n -= 50; }
    if (n >= 40) { strcat(numeral, "XL"); n -= 40; }
    while (n >= 10) { strcat(numeral, "X"); n -= 10; }
    if (n >= 9) { strcat(numeral, "IX"); n -= 9; }
    if (n >= 5) { strcat(numeral, "V"); n -= 5; }
    if (n >= 4) { strcat(numeral, "IV"); n -= 4; }
    while (n >= 1) { strcat(numeral, "I"); n -= 1; }

    /* If we made it here, we're OK. */
    return 0;
}
```

toRoman in C++

```
/*
 * This is the toRoman() function written in C++ --- note how it's
 * slightly different from the C version now.
 */
#include <string>
#include <stdexcept>

using namespace std;

string toRoman(int n) {
    if (n <= 0) { throw out_of_range("No Roman equivalent"); }
    string numeral;
    while (n >= 1000) { numeral += "M"; n -= 1000; }
    if (n >= 900) { numeral += "CM"; n -= 900; }
    if (n >= 500) { numeral += "D"; n -= 500; }
    if (n >= 400) { numeral += "CD"; n -= 400; }
    while (n >= 100) { numeral += "C"; n -= 100; }
    if (n >= 90) { numeral += "XC"; n -= 90; }
    if (n >= 50) { numeral += "L"; n -= 50; }
    if (n >= 40) { numeral += "XL"; n -= 40; }
    while (n >= 10) { numeral += "X"; n -= 10; }
    if (n >= 9) { numeral += "IX"; n -= 9; }
    if (n >= 5) { numeral += "V"; n -= 5; }
    if (n >= 4) { numeral += "IV"; n -= 4; }
    while (n >= 1) { numeral += "I"; n -= 1; }
    return numeral;
}
```

toRoman in Java

```
package roman;

/**
 * The Roman class houses the toRoman() method.
 */
public class Roman {
    /**
     * Returns the Roman numeral equivalent of the given integer.
     */
    public static String toRoman(int n) {
        if (n <= 0) { throw new OutOfRangeException("No Roman equivalent"); }
        String numeral = "";
        while (n >= 1000) { numeral += "M"; n -= 1000; }
        if (n >= 900) { numeral += "CM"; n -= 900; }
        if (n >= 500) { numeral += "D"; n -= 500; }
        if (n >= 400) { numeral += "CD"; n -= 400; }
        while (n >= 100) { numeral += "C"; n -= 100; }
        if (n >= 90) { numeral += "XC"; n -= 90; }
        if (n >= 50) { numeral += "L"; n -= 50; }
        if (n >= 40) { numeral += "XL"; n -= 40; }
        while (n >= 10) { numeral += "X"; n -= 10; }
        if (n >= 9) { numeral += "IX"; n -= 9; }
        if (n >= 5) { numeral += "V"; n -= 5; }
        if (n >= 4) { numeral += "IV"; n -= 4; }
        while (n >= 1) { numeral += "I"; n -= 1; }
        return numeral;
    }

    /**
     * Exception helper.
     */
    public static class OutOfRangeException extends RuntimeException {
        public OutOfRangeException(String message) {
            super(message);
        }
    }
}
}
```

toRoman in JavaScript

```
/*
 * This is the toRoman() function written in JavaScript. Note how we look up
 * a table instead of making a lot of conditional decisions.
 */
function toRoman(n) {
    var roman_map = {
        1000: 'M', 900: 'CM', 500: 'D', 400: 'CD', 100: 'C', 90: 'XC', 50: 'L',
        40: 'XL', 10: 'X', 9: 'IX', 5: 'V', 4: 'IV', 1: 'I'
    };

    var roman_string = "";
    for (var number_key in roman_map) {
        while (number_key <= n) {
            roman_string += roman_map[number_key];
            n -= number_key;
        }
    }

    return roman_string;
}
}
```

toRoman in ML

```
(*
 * toRoman returns the roman numeral equivalent of its input. Raises an exception
 * if the input is non-positive.
 *)
local
  val symbols = [ (1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"),
                  (90, "XC"), (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I") ];

  (*
   * Helper: r n symbols result => returns the roman equivalent of n
   * appended to result, using only the translations in the mapping
   * called symbols.
   *)
  fun r 0 symbols result = result
    | r n [] s = raise Fail "Cannot happen"
    | r n (symbols as (value, rep) :: tail) result =
      if n >= value then
        r (n - value) symbols (result ^ rep)
      else
        r n tail result
in
  fun toRoman n =
    if n <= 0 then
      raise Fail "No Roman equivalent"
    else
      r n symbols ""
end;
```

toRoman in Perl

```
#
# This is the toRoman() algorithm written in Perl.
#
use strict;
use warnings;

#
# Returns a string representing the roman numeral for the positive integer passed in.
#
sub toRoman {
  # Snag the lone argument.
  my $n = shift;

  die "$n is not a positive integer" if ($n !~ /\d+$/);
  die "$n has no Roman equivalent" if ($n <= 0);

  my %roman_map = (
    1000 => 'M', 900 => 'CM', 500 => 'D', 400 => 'CD', 100 => 'C', 90 => 'XC',
    50 => 'L', 40 => 'XL', 10 => 'X', 9 => 'IX', 5 => 'V', 4 => 'IV', 1 => 'I'
  );

  my $roman_string = "";

  foreach my $number_key (sort {$b <=> $a} keys %roman_map) {
    while ($number_key <= $n) {
      $roman_string .= $roman_map{$number_key};
      $n -= $number_key;
    }
  }

  return $roman_string;
}
```