# University of Kansas High School Design Programming: Rules, Resources, and Examples

Engineering SELF Fellows, KU Association for Computing Machinery

November 5, 2014

## Contents

# 1 Introduction

Welcome to the High School Design Programming Competition! This competition has been designed to test your knowledge of programming as a tool for general problem solving in your choice of the following languages:

C
C++
Java
Python (v. 2.7 and 3.4)
Ruby

If you have questions regarding the competition, please feel free to contact:

Aleksander Eskilson
Computer Science Undergraduate
Programming Competition Director
aeskilson@ku.edu

## 2 Rules

You will be given a docket describing a total of 10 programming challenges, arranged in ascending difficulty, and you will be allotted three and a half hours, in a single block of time, to write solution programs for these challenges. An automated system called DOMJudge provides a website interface, http://bit.ly/YgPIPg or its full url, http://http://ec2-54-88-68-156.compute-1.amazonaws.com/, where you can submit your solutions for judging, receive feedback, and check a running scoreboard and clock. Before the actual day of the competition, a test competition will be available here. The login username: team, password: pass, will allow you to browse the interface and even submit a solution to the problems of the test contest. Here, you can test submitting your programs for automated judging.

All problems will be described in a uniform fashion: An introduction to the problem will set up the scenario and describe what challenge you must write a program to solve. It will then describe how the input your program must read will be structured, and it will prescribe how the program's output must be formatted in order to be read correctly. An example of input and output will be provided for you to check your program against. **Note**: when your program is actually judged, different input data will be used, but the formatting will be the same. Pay very special attention to the section describing how to properly write your programs to accept input from DOMJudge.

Each correct problem is worth 1 point, so whichever team completes the most problems out of 10 will be designated the winner. In the case of ties, time (as marked in seconds, running from the beginning of the competition) will be used as a tiebreaker. So if more than 1 team completes all 10 problems, the first team to have done so is the winner. You may submit your solutions for judging as many times as you need to, but you should be aware that each time you submit a solution that is judged to be incorrect, you will receive a time penalty that will add to your total time, so it is best to work hard to ensure your program is correct before submitting for judging. However, even if your submission turns out to be wrong, you should feel free to rework it until you get it right or decide to move on to another problem.

Teams may consist of up to two students. You will not be allowed use of the Internet except for access to the DOMJudge web interface, but any books on your programming language may be used during the competition. Day of the competition, your team will be allowed one computer in the Self Computing Commons of Eaton Hall. 42 computers will be available, meaning we can support up to 42 teams. Half of those machines run Windows 7, the other half run Fedora Linux. Software which students can use to write and test their programs will be uniform across both types, and submission to the DOMJudge system for judging is done through an Internet browser, so the students competitive experience will be uniform on both machines. You may make a request for your Operating System of choice however, but choices will be appropriated on a first-come registration basis.

# 3 Concepts

In this section, we list the core programming concepts you should be familiar with in your langauge(s) of choice. All the problems day-of the competition will be solvable using just the concepts mentioned here. However, as the problems get more difficult, solutions will take less time to draft, code, and test if your team is well practiced and quite familiar with your chosen languages. Although we list here the basic concepts necessary to compete, we encourage you to read more in depth about features of your language, so that you may be more creative and efficient in writing your solutions. You may already be familiar with the basic stuff. If so, treat what follows as a refresher, then get to practice! In our little example snippets below, the code is Java.

## 3.1 Assignment and Arithmetic

All languages have some manner of giving names to data and doing basic operations on that data. You should be familiar with how your language declares variables and assigns values to them. You should also know how your language performs arithmetic operations, like addition and multiplication. In particular, you will want to be familiar with 'Modular Arithmetic', an operation that works by finding the remainder when you divide one integer number by another. Other useful math functions are Ceiling and Floor, which round decimals up or down to the nearest integer.

You should also know about all the different 'primitive' data types your language has. Most languages can natively represent strings of characters, single characters, Boolean values (true and false), integers, and decimal numbers. But for some languages, 3 / 4 may not be the same as 3 / 4.0. Knowing little quarks about your language keeps you from running into odd errors.

## 3.2 Conditionals

Conditional branches allow you to evaluate logic – in particular, they allow your program to change the instructions it will execute depending on the kind of data your program was given. The most common way to represent conditional choice in a program is with `if else` statements. In general, when your program reaches a conditional block of code, it determines whether or not the condition following the `if` is true. For example:

```
1 int x = 4;
2 if (x == 5) {
3     System.out.println("x is five!");
4 } else {
5     System.out.println("x is not five, bummer");
6 }
```

Here, we let our integer x be 4. When we reach our conditional, we check if x equals 5. If it does, we'd print out "*x is five!*", but in our case, x does not equal 5, so instead we would evaluate the `else` portion and print "*x is not five,*

*bummer*". Most languages will also let you chain several `if else` statements together,

```
1 String ourSentence = "Coi ro do!";
2 if (ourSentence.equals("Hi there!")) {
3     System.out.println("Hi right back at ya!");
4 } else if (ourSentence.equals("I have to go..")) {
5     System.out.println("See ya later!");
6 } else {
7     System.out.println("Sorry, I don't speak Lojban!");
8 }
```

Above, we use Java's special `equals` method to check if a string is the same as another string. So there are lots of things that can be used to create a conditional statment in an `if else` block. And on top of that, you can put together several conditionals using logical operators || for *or* and `&&` for *and*. For example, `if (x * 2 == 4)||(x >= 10)` reads as *if x times 2 is equal to four or, if x is greater than or equal to 10....*

So for conditionals, you should know how your language writes `if else` blocks, what methods or operations can check if something is *true* or *false*, and how conditionals can be linked together using *and* and *or*.

## 3.3 Loops

Loops let you repeat a block of code again and again either a certain number of times, or until some condition (remember those?) is met. The basic loops for Java, C, and C++ are `for`, `while`, and `do while`. Python and Ruby have loops named the same, but they look and work slightly different. Review for your language what loops you have and how they work.

## 3.4 Data Structures

When we say *Data Structures*, we really just mean containers. Common containers in Java, C, and C++ include Arrays, Lists (or Vectors), and Hashmaps. Arrays and Lists are the simplest. Arrays are like a line of cubby holes, they have fixed length, and can be filled up with only one type of data at a time, like an *Array of integers* or an *Array of Strings*. Lists work similarly, but can grow in length. Both of these structures let you check what value was assigned to a particular index, and they let you assign values to indices. Hashmaps (also sometimes called Dictionaries) let you give pair a value to a keyword so that you can look up values if you know their keywords, or you can check if a value is in the hashmap. Arrays, Lists, and Hashmaps all have different ways of being used, and are useful at different times. You should be familiar with what Data Structures your language has and the ways they can be used. Arrays and Lists though will be your bread and butter for most problems at the competition.

**Note:** In Python and Ruby, Lists are usually preferred to Arrays for the sake of simplicity.

## 3.5 Functions and Methods

All languages have special commands that either return specific pieces of information, or transform data in some way. For example, in Java, if we want to know the length of a string in the variable `ourString`, we could use `int stringLength = ourString.length();`, which would give an integer representing the length of the string. In Python, if we wanted to know the size of a list `ourList`, we could use `listLength = len(ourList)`. All the methods and functions of a language are described by the language's *Standard Library*. The list of total methods and functions in the Standard Library is often quite large, but it'll be fine if you just get familiar with those related to your language's data structures and Strings.

Most languages will also let you define your own methods and functions elsewhere inside your program. This can help your code keep clean if you find yourself repeating some pattern of operations multiple times, and knowing how your language declares and uses them may be helpful.

In particular, you will want to be familiar with methods or functions in your language that work with strings and arrays (or lists for Python and Ruby teams).

## 3.6 Going Forward

To best learn and practice the concepts listed above, we recommend you check out the websites `http://www.codecademy.com` or `http://www.codingbat.com`. They'll give you lots of simple challenges and tutorials that will introduce you to the languages supported by the competition. Each of them have lots of examples and even videos to help you learn the basic concepts in writing programs. **Note:** although these resources will help you learn aspects of your chosen language, we strongly encourage you to install an IDE (Integrated Development Environment) and grow familiar with how to write programs using it. Feel free to log into the test competition and submit test solutions for the problems included there.

For each language supported by the competition, there are several popular IDEs, and the computers provided to you to write and submit your solutions will have many available. The most common is Eclipse, which has great support for Java and C/C++. For Python, there's Idle, and for Ruby, there's Aptana. All of these IDEs are freely available. Naturally, you should use whichever you feel most comfortable with. If you would like to ensure your IDE or text-editor will be available to you, feel free to email the Programming Competition director, whose email is on the second page of this packet. We'll see if your requested IDE is already installed, or if it can be.

# 4 DOMJudge

## 4.1 Summary

Here we include a short summary of the DOMJudge system interface, the web-portal by which you will submit your solutions for automated judging. The single constraint of this system is that your programs **must** be written to accept their input from `stdin` and write their output to `stdout` (also sometimes referred to as *writing to the console*). We talk a little more about what this means in the examples portion of this section. What follows is an introduction to the interface.

## 4.2 Interface

A web URL will be provided to you from which you will gain access to the DOMJudge submission interface. From your team pagei, `http://example.com/domjudge/team`, you will select **Select file...** in the left column and select the file of your source code to submit. By default, the problem is selected from the base of the filename and the language from the your file's file extension.

Viewing scores, submissions, and sending and reading clarification requests is done through the web team interface, `http://example.com/domjudge/team`.

The left column of the team interface shows your teams's row in the scoreboard: your position and the which problems you attempted and which you solved. Via th menu, you can view the public scoreboard page with scores of all the teams. The score column lists the number of problems solved and the total penalty time. Each cell in the problem column lists the number of submissions, and if th problem was solved as well as the time of the first correct submission for that problem since the contest start. Submissions can have the following resuts:

**CORRECT:** The submission passed all tests: you solved this problem!

**COMPILER-ERROR:** There was an error when compiling your program. On the submission details page you can inspect the exact error (this option might be disabled).

**TIMELIMIT:** Your program took longer than the maximum allowed time for this problem. Therefore it has been aborted. This might indicate that your program hangs in a loop or that your solution is not efficient enough.

**RUN-ERROR:** There was an error during the execution of your program. This can have a lot of different causes like division by zero, incorrectly addressing memory (e.g. by indexing arrays out of bounds), trying to use more memory than the limit, etc. Also check that your program exits with exit code 0!

**NO-OUTPUT:** Your program did not generate any output. Check that you write to standard out.

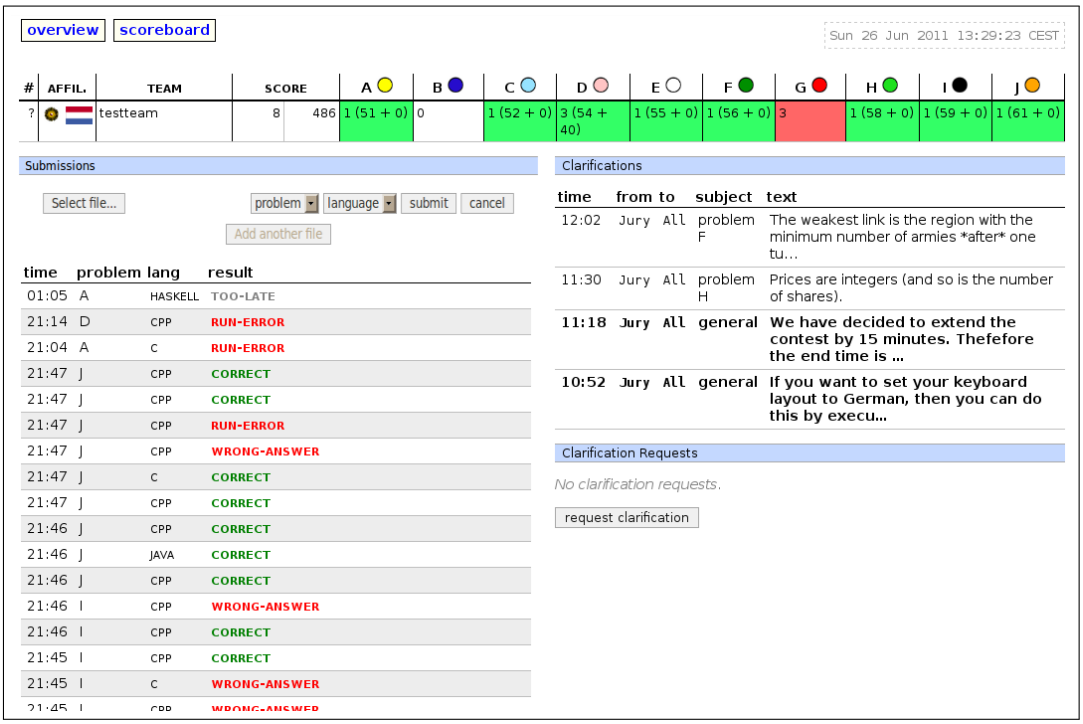


Figure 1: the web team interface overview page

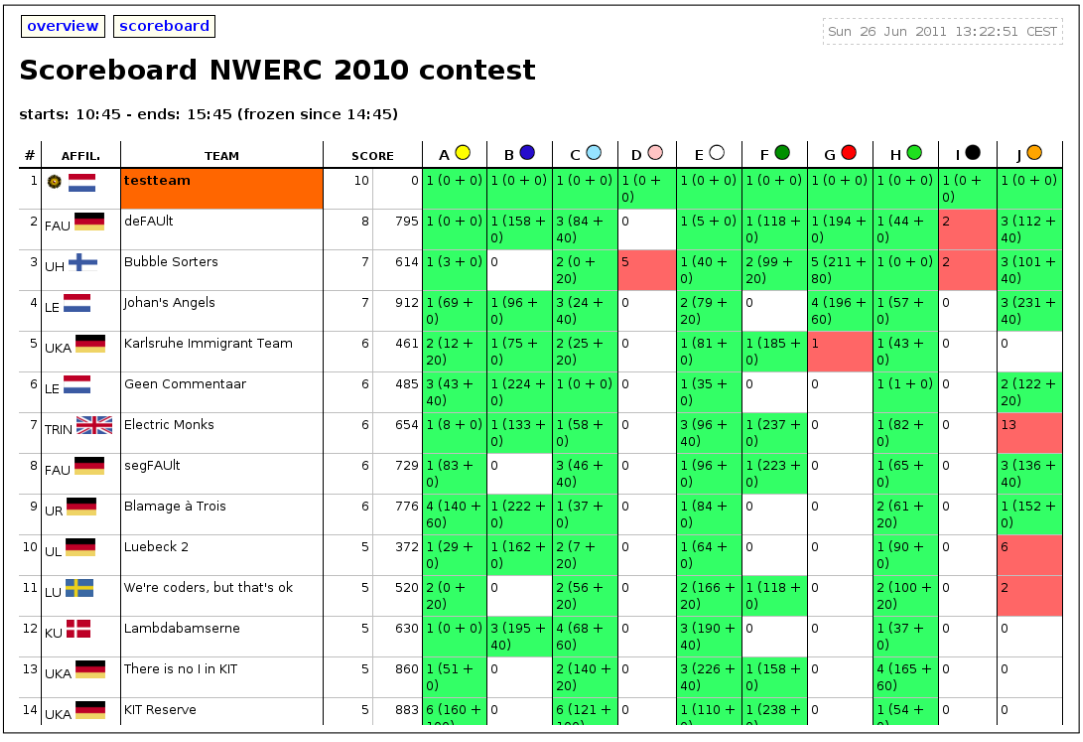


Figure 2: the scoreboard webpage

**WRONG-ANSWER:** The output of your program was incorrect. This can happen simply because your solution is not correct, but remember that your output must comply exactly with the specifications of the jury.

**PRESENTATION-ERROR:** The output of your program has differences in presentation with the correct results (for example in the amount of whitespace). This will, like WRONG-ANSWER, count as an incorrect submission. This result is optional and might be disabled.

**TOO-LATE:** Bummer, you submitted after the contest ended! Your submission is stored but will not be processed anymore.

You are allowed to communicate with the jury, we omniscient few who oversee the entirety of the competition namelessly, and behind the inscrutability of teletype terminals, through the use of clarifications. These can be found in the right column of your team page. Both clarification replies from the jury and requests sent by your are displayed there.

There is also a button to submit a new clarification request to the jurry. This request is only readable for the jury and they will respond as soon as possible. If the jury deems an answer relevant to everyone, it will be viewable to everyone.

## 4.3 Testing

The DOMJudge system is fully automated. Once input/output files have been prepared to verify your program, no other human element is necessary in scoring besides clarifications on the part of the jury. But this is how it works:

With the web interface you can submit the source code of your solution to the jury. Note that you must send your source code, not a compiled version of the program. Your solution is then sent over the network to a judging computer that will automatically compile or interpret your solution and pipe the proper input file to the program. The output of your program will then be compared against the validated output file. If your output matches the expected output, your program is correct. If not, you will receive some information about what went wrong before you go back to the drawing board.

## 4.4 Examples of Proper Input/Output

Here we define how you must write your program to read from `stdin` and write to `stdout` for the languages supported by this competition. We do this through annotation of two example programs. The purpose of this section is to expose you to the common input/output functions of your language that will allow your program to accep intput from DOMJudge and write output back to it.

**Description:** Write a program to send a friendly greeting!

**Input:** Write a program to read a list of names headed by an integer from `stdin`. The first line will be an integer representing the number of lines to read. Each subsequent line will be a single name.

**Output:** On a new line for each, print out to `stdout` the name you read as part of the string "Hello <name>!".

**Sample:**

| **input** | **outout** |
|---|---|
| `3`<br>`World`<br>`Alan`<br>`Ada` | `Hello world!`<br>`Hello Alan!`<br>`Hello Ada!` |

A solution for the problem in C:

```c
#include <stdio.h>

int main() {
    int ntests;
    char name[100];

    scanf("%d\n", &ntests);

    int i;
    for (i = 0; i < ntests; i++) {
        scanf("%s\n", name);
        printf("Hello %s!\n", name);
    }

    return 0;
}
```

Here, we note that C uses the `scanf` function to read from `stdin` and `printf` to write to `stdout`. It declares an integer variable `ntests`, and `scanf` reads in a value from the console and assigns that to `ntests` (we note that `scanf` really uses a *reference* to `ntests`, that's what the *and* sign means, but that's special C stuff). `scanf` is later used with a character array that will hold our names. Our character array's size is 100, which is just arbitrarily large so we can hold any name we might read. We then use the `printf` function to print out our strings to `stdout` as we loop through them. We have to always `return 0` or else our program will **error**!

A solution in C++:

```cpp
#include <iostream>
#include <string>

using namspace std;

int main() {
    int ntests;
    string name;

    cin >> ntests;
    for (int i = 0; i < ntests; i++) {
        cin >> name;
        cout << "Hello " << name << "!" << endl;
    }

    return 0;
}
```

C++ uses `cin >>` to push `stdin` lines into a variable on the right, and uses `cout >>` to print out lines from the right to `stdout`. C++ also has real strings, not

just character arrays. Other than that, the logic of this solution is the same as the solution in C.

A solution in Java:

```
1  import java.io.*;
2
3  class Main {
4      public static BufferedReader in;
5
6      public static void main(String[] args) throws IOException {
7          in = new BufferedReader(new InputStreamReader(System.in));
8
9          int nTests = Integer.parseInt(in.readLine());
10
11         for (int i = 0; i < nTests; i++) {
12             String name = in.readLine();
13             System.out.println("Hello " + name + "!");
14         }
15     }
16 }
```

For Java, we first note our class **needs** to be named *Main*. Java uses a `BufferedReader` object that it imports from `java.io.*`, which it uses to gather input from `System.in`. Using the command `System.out.println(...)` will let you write to `stdout`.

A solution in Pythoni 2.7:

```
1 import fileinput
2 import sys
3
4 num = sys.stdin.readline()
5
6 for line in fileinput.input():
7     print "Hello " + line.strip() + "!"
```

For Python 2.7, we import `fileinput` and `sys` to get our input from `stdin`. Python has some looping mechanisms that will allow us to cleverly not need the integer representing the number of lines, the command on line 4 reads it in, but does nothing with it. The special Python `for` loop on line 6 will loop over all the lines from `stdin` and store each one temporarily in the variable `line`. It's worth noting that if we read in the lines this way, we need to strip off the hidden `newline` character that separates each line of input from the string we've read in. `strip()` does that for us. Then we just print it out to `stdout` with the aptly named `print` function.

A solution in Python 3:

```
import fileinput
import sys

num = sys.stdin.readline()

for line in fileinput.input():
    print("Hello " + line.strip() + "!")
```

Python comes in two common versions. Python 2.7 and Python 3 are just different enough that it matters. All we notice here is that the Python 3 `print` command wraps its contents in parenthesis. There are some other differences too, so make sure you know which version of Python you're using.

**Note:** for both versions of Python, it's common to include *interpreter directives* at the top of the file, that may look like `#!/usr/bin/python`. You must **not** include a line like this in your submission. DOMJudge will be mad at you. If this doesn't look familiar to you anyway, just forget about it and go on with your day.

A solution in Ruby:

```
\$stdin.each_line.each_with_index do |line, i|
    if i == 0
        next
    end
    puts "Hello #{line.strip}!"
end
```

Ruby uses a special variable `$stdin` to represent console input. The method `each_line` grabs all the lines from `stdin` at one time, and the method `each_with_index` allow us to keep track of our position in the loop. Since the first line is just a silly integer, we can use the index of 0 to skip it, then we can do what we'd like with subsequent lines. In our case, we use the function `puts` to write to `stdout`. You'd likely replicate this looping pattern if you decide to use Ruby.

**Description:** Write a program to simplify fractions.

**Input:** Write your program to read pairs of integers from `stdin`. On each line of input, there will be a pair of space separated integers. The first will represent the `numerator` of the fraction, the second will represent the `denominator`. The end of input will be signaled by a `0`. For a pair of integers, you will never be given a numerator or denominator with a value of `0`. The only `0` given as input will be the one that signals the end of input.

**Output:** Write to `stdout` the simplified fraction, with the numerator separated from the denominator by a *forward slash*, `/`. Each simplified fraction should be separated from the next by a newline. If the simplified fraction would be a single integer, write it without a denominator.

**Sample:**

| **input** | **outout** |
|---|---|
| `18 12`<br>`28 49`<br>`99 11`<br>`13 5`<br>`0` | `3/2`<br>`4/7`<br>`9`<br>`13/5` |

## 4.5 Going Forward

Now that you know how DOMJudge expects you to use Input/Output, feel free to log into the guest account of our test DOMJudge instance and review the practice problems there. You can also get a feel for the environment you'll use to judge submit your programs. Additional practice problems and even examples of solutions can be found at the github repository of the KU chapter of the Assocation for Computing Machinary, https://github.com/KU-Competitive-Programming/Weekly-Problems. Here we keep a backlog of interesting challenges and solutions our members have submittted in the past. Any problem headed *Beginner* should be good for you to read the description of.