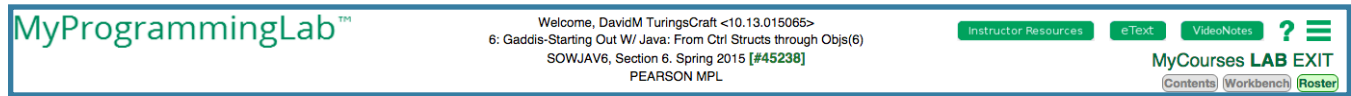


## THE MyProgrammingLab (MPL) INTERFACE: QUICK REVIEW/OVERVIEW

There are five primary elements in the MPL interface:

The main page **header** is always available at the top of the page and provides important product identification and interface elements that are relevant in all application modes.



**Boxes** are virtual windows within the web page-- they are resizable, draggable and can be made visible or invisible by clicking their corresponding place-holders in the header. There are three boxes:

*Table of Contents (TOC)*

*Workbench*

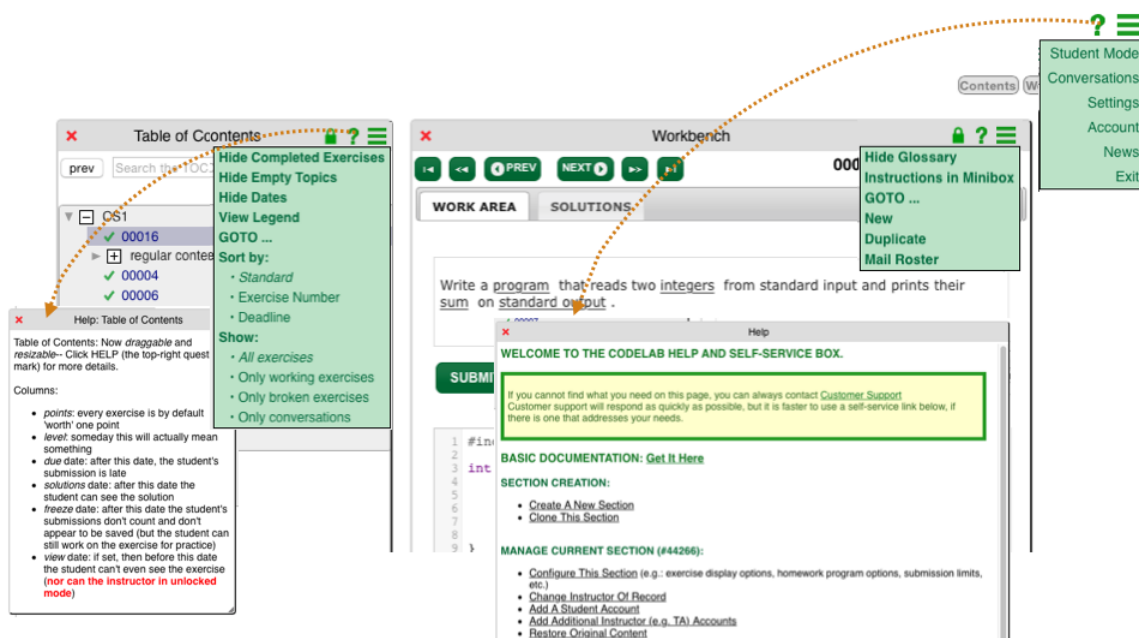
*Roster*

The heading area of each box contains an **X** for hiding the box, a **?** for obtaining box-specific assistance, and a 3-bar icon **≡** for box-specific actions and options. In addition, if the user has permission to edit the contents of the box, there is a lock icon **🔒** that can be locked/unlocked. When unlocked, the header gets an additional element: **Save**

**Miniboxes** are temporary boxes that appear automatically in response to a user action and disappear under user control. They display data and in some cases controls that relate to a user action that triggered their appearance. Miniboxes are typically resizeable and draggable and usually just have one control: an **X** for closing the minibox. Once closed, the minibox is gone until the action that triggered its appearance is repeated.

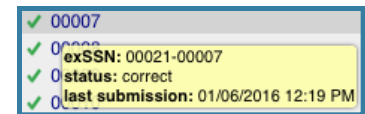
Mouse-over generated "**tooltips**" contain expanded information about a particular data element. They are very ephemeral and come and go with mouse-over operations.

**Popup windows:** a few capabilities, for example new section creation, have not yet been integrated into the Box/Minibox system. Until they are, they will appear in completely separate browser windows (or tabs).



## ESSENTIAL FACTS ABOUT EXERCISES

**Identification:** Every exercise is uniquely identified by an *exercise number*, often called an "exercise-ssn" or "exSSN". The full exercise-ssn consists of an **internal user ID and a sequence number for that user**. For example, 00021-00007 is the 7th exercise created by user 21. Mousing over an exercise entry in the TOC reveals a tooltip that shows its exSSN.



**Ownership:** Every exercise is owned by somebody: either the instructor who created it, or by Turing's Craft. The owner is identified by the user ID in the exSSN. Turing's Craft exercises have a user ID of 0.

*How to tell if you own an exercise:*

select an exercise in the TOC

if there is a lock icon in the Workbench box header, it's yours

**Types:** There two kinds of exercises:

**Coding:** the student submits code which is compiled and executed with the outcome being examined to determine correctness and possibly offer guidance. The different types of coding exercises are discussed later in this document.

**Traditional:** multiple choice and fill-in exercises

## WHERE DO EXERCISES COME FROM?

New exercises are created by cloning exercises that already exist. There are two ways to do this:

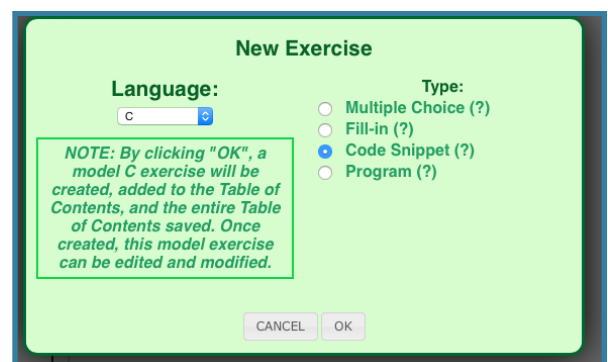
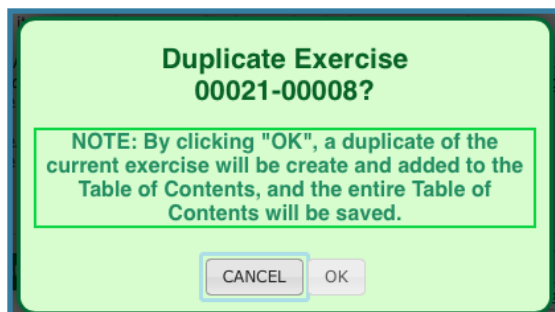
(1) *Duplicate Your Own Exercise:* Select an exercise that you own. (If a lock icon appears in the Workbench header, you own the exercise.) Mouse over the 3-bar icon of the workbench box and click **Duplicate**. (See the picture on the right.)



(2) *Duplicate A Model Exercise:* Mouse over the 3-bar icon and select **New**. (See the picture on the right.)



In both cases, a dialog box is displayed (see below).



In the second case ("New Exercise"-- that is duplicating a model exercise), the language and exercise type must be selected before confirming **OK**.

Once the confirmation OK is clicked:

*Cloning*: the exercise-- either the Turing's Craft model or the instructor's exercise-- is cloned

*Ownership*: the owner of the clone is the current instructor

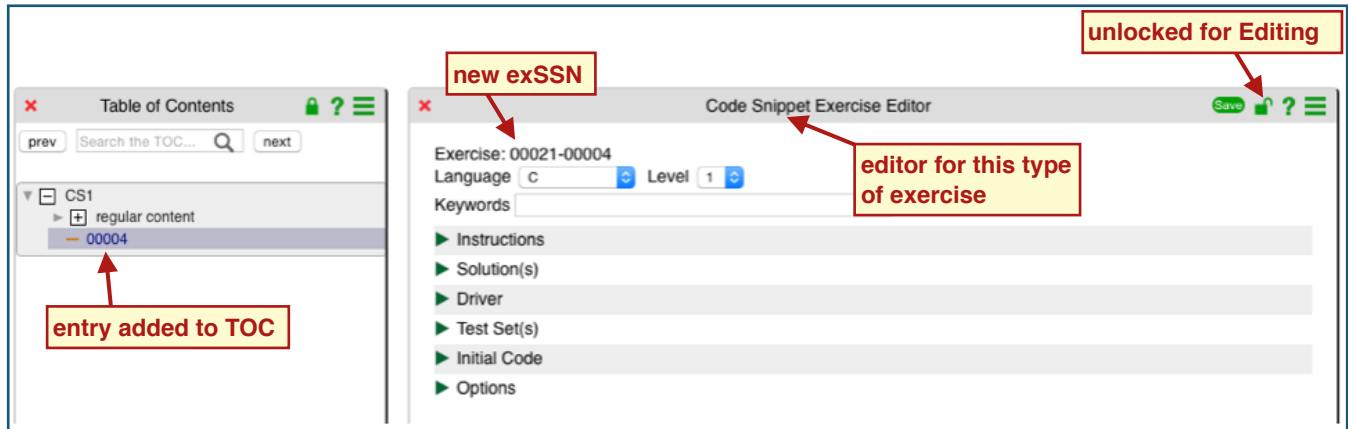
*exSSN Assignment*: a new exercise number is created and associated with the new clone

*Saving the exercise*: the new exercise is saved

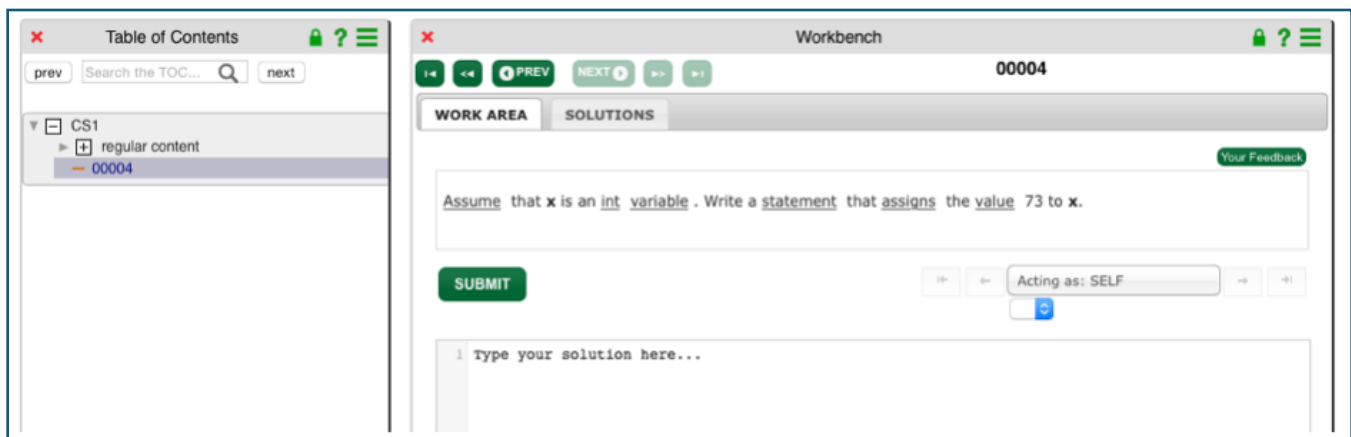
*Inserting a TOC entry*: an entry for the exercise is inserted into the Table of Contents

*Saving the TOC*: the Table of Contents is saved

*Unlocking*: the new exercise is unlocked to permit editing



At this point there is now a new, working exercise with an entry in the TOC. If the instructor were to log out or simply close the browser, the exercise would be there upon logging in and loading the course again:



Of course, all this just creates copies of existing exercises.

The point is to change the clone to be the intended new exercise.

To do that, it is necessary to understand the way exercises are specified.

We start with the Coding exercises.

## EXERCISE SPECIFICATION: PROGRAM EXERCISES

The interface for specifying coding exercises varies very little (see the picture on the right). The basic elements are:

**Language:** Choose the programming language.

**Level:** Choose a level-- currently the level has no standard meaning and it is currently simply a label of sorts

**Keywords:** An optional comma-separated list of "keywords" of the exercise author's choosing. Currently, they chiefly serve as a label, but can be searched for in the TOC and are also used to categorize the exercise in the REPOSITORY if the exercise is listed there

**Instructions:** The specification of the instructions-- an HTML editor toolbar is provided.

**Solutions:** There must be at least one solution provided.

**Driver:** For PROGRAM exercises this is generally empty.

**Test Sets:** These specify the external test sets applied to the student submission. There must be at least one Test Set provided. A test set consists of:

- specification of command-line arguments
- specification of input (standard input and any data files used as input)
- specification of output (standard output and any data files used as output)

**Initial Code:** generally empty, but if populated, its content is provided to the student from the beginning-- this supports exercises of the form: "modify the following program so that it ..."

**Options:** To be discussed later.

Language  Level   
 Keywords   
 ▶ Instructions  
 ▶ Solutions  
 ▶ Driver  
 ▶ Test Sets  
 ▶ Initial Code  
 ▶ Options

The crux of a Program Exercise specification lies in the specs for Instructions, Solutions, and Test Sets.

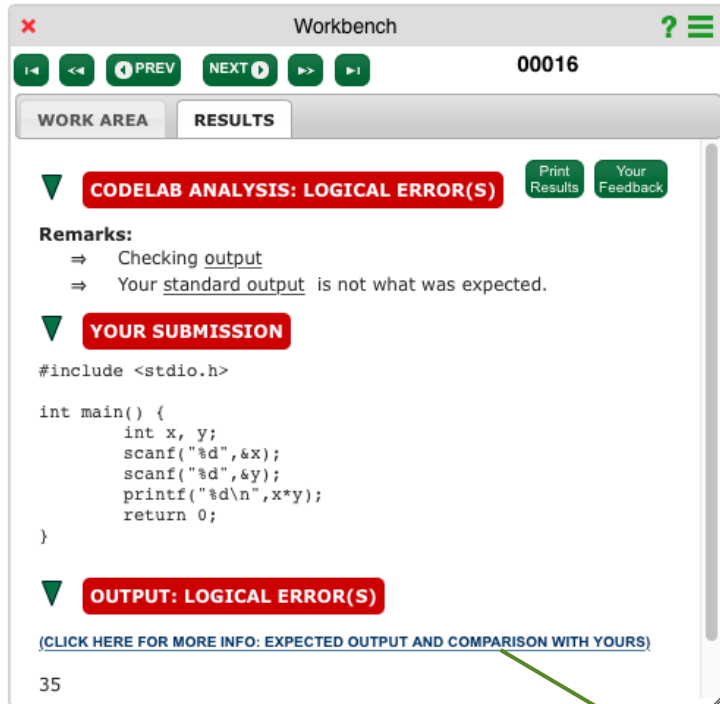
**A Simple Example.** Let's start with a simple example: a C program that reads two integers and prints their sum. To a student this exercise might appear thus:

The Workbench interface shows a student's submission. The top bar includes navigation buttons (PREV, NEXT) and a problem ID (00016). The main area contains the problem instructions: "Write a program that reads two integers from standard input and prints their sum on standard output." Below the instructions is a "SUBMIT" button and a text area for the solution. A green box labeled "submitting a correct solution" points to the submission process. The resulting analysis shows a green banner "CODELAB ANALYSIS: CORRECT!" and a "YOUR SUBMISSION" section displaying the submitted C code:

```
#include <stdio.h>

int main() {
    int x, y;
    scanf("%d", &x);
    scanf("%d", &y);
    printf("%d\n", x+y);
    return 0;
}
```

Suppose the student used \* instead of +:

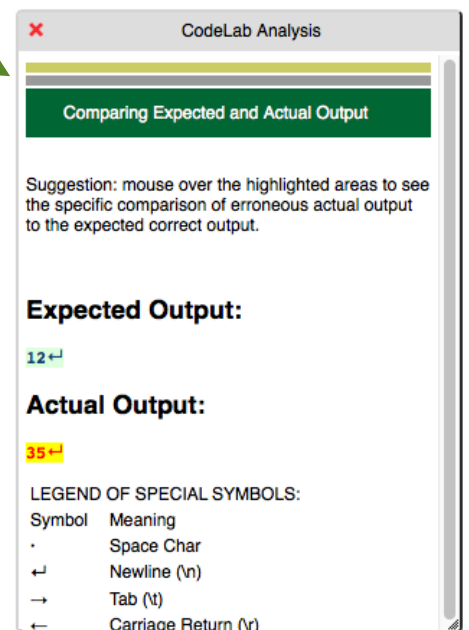


The screenshot shows the 'Workbench' window with a 'RESULTS' tab. It displays a 'CODELAB ANALYSIS: LOGICAL ERROR(S)' message. The 'Remarks' section indicates that the 'standard output' is not what was expected. The 'YOUR SUBMISSION' section shows a C program that calculates the product of two numbers instead of their sum. The 'OUTPUT: LOGICAL ERROR(S)' section contains a link: '(CLICK HERE FOR MORE INFO: EXPECTED OUTPUT AND COMPARISON WITH YOURS)'. The line number 35 is visible at the bottom left.

```
#include <stdio.h>

int main() {
    int x, y;
    scanf("%d", &x);
    scanf("%d", &y);
    printf("%d\n", x*y);
    return 0;
}
```

Clicking the link at the bottom yields a minibox with a helpful comparison between the correct expected output and the student's incorrect output.



The 'CodeLab Analysis' minibox is titled 'Comparing Expected and Actual Output'. It provides a suggestion to mouse over highlighted areas for a specific comparison. It shows the 'Expected Output' as '12' and the 'Actual Output' as '35'. A legend of special symbols is provided at the bottom.

**Expected Output:**  
12

**Actual Output:**  
35

**LEGEND OF SPECIAL SYMBOLS:**

Symbol	Meaning
·	Space Char
↵	Newline (\n)
→	Tab (\t)
↵	Carriage Return (\r)

**Specification.** The specification for this exercise is shown on the right.

**How It Works: Save-Time.** When the instructor clicks **Save**, the system iterates over each provided solution (only one in this specification) and for each provided test set (only one in this specification) it:

**Compiles:** the solution is compiled to generate an executable file.

**Execution Preparation:** each of the named input files (none in this case) are created, named and populated according to the input files in the specification. A special input file is created to hold the specified standard input (7 and 5 in this case).

**Execution:** the executable file is invoked with the specified command-line arguments (none in this specification) with standard input redirected from the aforementioned special input file. Standard output is redirected to a special output file.

**Analysis:** if the execution did not end with a status code of 0, the specification is deemed to be flawed and the **Save** fails. If the execution ends with a status code of 0 the specified output files are compared with the ones actually generated. If there is a difference, then the **Save** fails.

In other words, the system makes sure that the instructor-provided solution can pass the instructor-provided tests.

**How It Works: Submit-Time.** When the student submits code, the above process is carried out but instead of iterating through the instructor solutions, the student code undergoes the Compile/Execution Preparation/Execution/Analysis process.

The screenshot shows the 'Classic Program Exercise Editor' window. At the top, it displays 'Exercise: 00021-00016', 'Language: C', and 'Level: 1'. Below this is a 'Keywords' field. The 'Instructions' section contains a text box with the instruction: 'Write a program that reads two integers from standard input and prints their sum on standard output.' The 'Solutions' section shows a code editor with the following C code:

```
#include <stdio.h>

int main(int ac, char *av[]) {
    int x, y;
    scanf("%d", &x);
    scanf("%d", &y);
    printf("%d\n", x+y);
    return 0;
}
```

The 'Test Sets' section is expanded, showing 'Command-line Arguments' with an empty 'Args' field. Under 'Input', 'Stdin' is selected, and the 'Data' field contains '7 5'. Under 'Output', 'Stdout' is selected, and the 'Data' field contains '12'. At the bottom, there are checkboxes for 'Ignore Whitespace', 'To Single Whitespace', 'Ignore Case', and radio buttons for 'Plain Text: no regular expressions', 'Regex: metachars are literal unless escaped', and 'Regex: metachars are special unless escaped'. The 'Initial Code' and 'Options' sections are collapsed.

**A More Complicated Example: the standard Model Program exercise.** Now let's turn to the Program exercise that the system clones when a NEW program exercise is created. Computationally, it is as simple as the previous example. What's different is that it involves command-line arguments, reads multiple input files, and writes output to a file instead of standard output. The instructions are:

Write a program that gets 2 or more command-line arguments. The last argument is a file that the program will send its output to. The other arguments are names of input files that already exist and that contain integers.

The program opens each of the existing input files, in the order given by the arguments, and reads and sums all the integers of each file, writing the name of the input file followed by its sum to the output file (given by the last argument).

EXAMPLE: If the program were invoked like this: `a.out fileA fileB fileZZZ` and if the content of fileA was "3 9 8" and that of fileB "4 1" then the output written to fileZZZ would be:

```
fileA 20
fileB 5
```

The system's solution for this is:

```
#include <stdio.h>

void readSumAndWrite(char *fname, FILE *fpout) {
    int n, result, total=0;
    FILE *fpin = fopen(fname,"r");
    result = fscanf(fpin,"%d",&n);
    while (result!=EOF) {
        total+=n;
        result = fscanf(fpin,"%d",&n);
    }
    fprintf(fpout,"%s %d\n", fname, total);
    fclose(fpin);
}

int main(int ac, char *av[]) {
    FILE *fpout;
    int i;
    fpout = fopen(av[ac-1],"w");
    for (i=1; i<ac-1;i++)
        readSumAndWrite(av[i], fpout);
    return 0;
}
```

The system's solution for this is:

The specification for this is shown on the next page.



The full set of instructions and the solution for this exercise were shown above.

Again there is only one test set.

Note that there are two input files in the test set, but only one shown here. The other input file is named "fileB" and contains the values "4 1".

**How It Works: Save-Time:** the same way as in the simple example above.

**How It Works: Submit-Time:** the same way as in the simple example above.

The screenshot shows the 'Classic Program Exercise Editor' window. It has a title bar with a close button, a 'Save' button, and help icons. The main content is organized into several sections:

- Instructions:** Contains a rich text editor with a toolbar (undo, redo, bold, italic, text color, background color, link, unlink, image, table, list, link, source) and a text area with the following text:
 

```
Write a program that gets 2 or more command-line arguments.
The last argument is a file that the program will send its output to.
The other arguments are names of input files that already exist
and that contain integers.
```
- Solutions:** Contains a code editor with the following C code:
 

```
#include <stdio.h>

void readSumAndWrite(char *fname, FILE *fpout) {
    int n, result, total=0;
    FILE *fpin = fopen(fname, "r");
    result = fscanf(fpin, "%d", &n);
    while (result!=EOF) {
        total+=n;
    }
}
```
- Driver:** A section for the driver code.
- Test Sets:** Contains a 'Command-line Arguments' section with an 'Args' field containing 'fileA fileB fileZZZ'. Below it is an 'Input' section with radio buttons for 'Stdin' and 'File' (selected), and a text field containing 'fileA'. There is also a 'Data' section with a text area containing '3' and '9 8'. Below that is an 'Output' section with radio buttons for 'Stdout', 'Stderr', and 'File' (selected), and a text field containing 'fileZZZ'. There is also a 'Data' section with a text area containing 'fileA 20' and 'fileB 5'. At the bottom of the test sets section are four checkboxes: 'Ignore Whitespace' (checked), 'To Single Whitespace', 'Ignore Case', and 'Plain Text: no regular expressions' (selected). Below these are two radio buttons for 'RegEx: metachars are literal unless escaped' and 'RegEx: metachars are special unless escaped'.
- Initial Code:** A section for initial code.
- Options:** A section for options.



## USING THE EXERCISE EDITOR INTERFACE

The interface reflects the nested logical structure of an exercise specification.

An *exercise* is:

- 1 set of instructions
- 1 or more solutions
- 1 driver
- 1 or more test sets
- 1 set of initial code
- 1 set of options

and a *test set* is:

- 1 string of command-line arguments
- 1 or more input sources
- 1 or more output targets

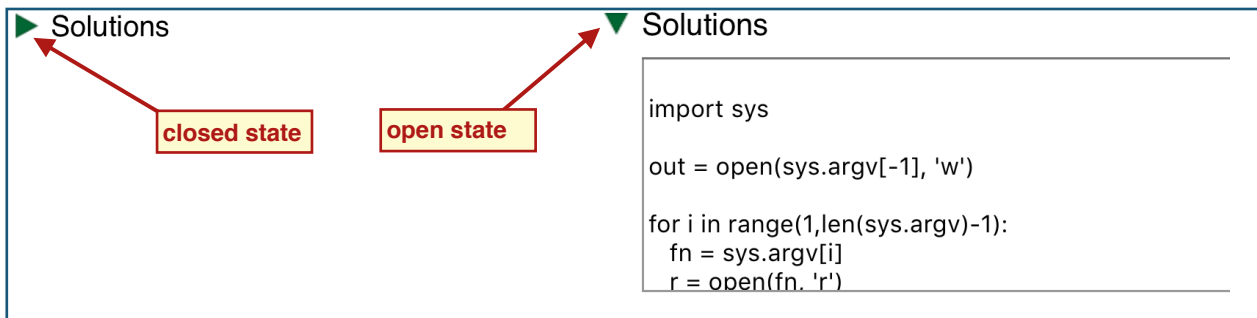
and an *input source* is

- a file identification (stdin or a filename)
- a set of data

and an *output target* is

- a file identification (stdout or stderr or a filename)
- a set of data

Every composite element (i.e. a category containing form elements) is named and is preceded by an open/close green triangle control: ▼/▶.



Those composite elements of which there can be more than one (e.g. solution, test set, input source, output target) have, in addition, a set of controls that permit the addition of a new empty element, duplication of the current element or deletion of the current element: **New** **Dup** **Del**. Such elements are only displayed one at a time. Next to the control is an indication of which element is being displayed (out of how many): 1 of 1, 1 of 2, etc. Thus:

**New** **Dup** **Del** 1 of 1

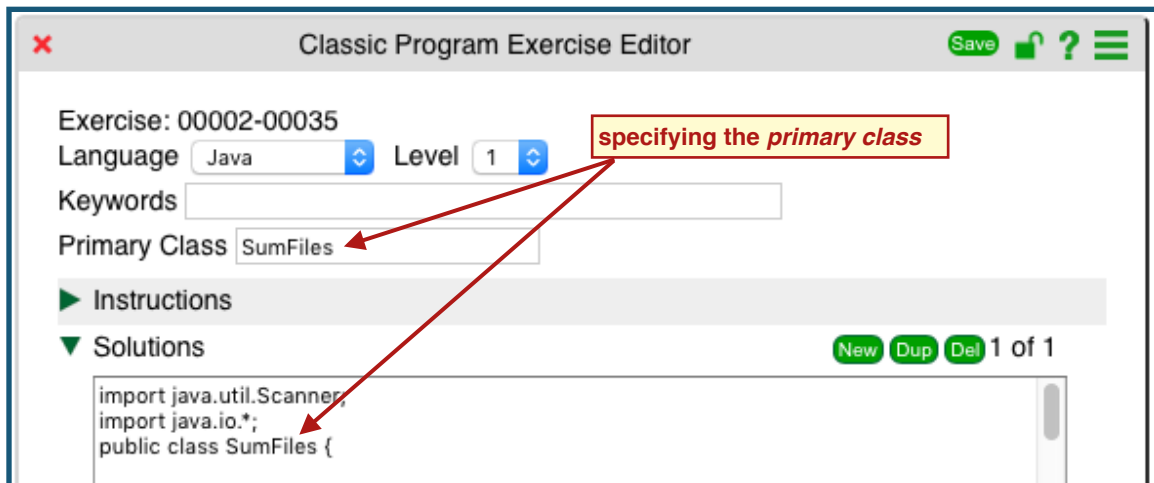
when there is more than one element, arrow navigation controls are displayed, and these can be used to move from one element to the next:

**New** **Dup** **Del** ◀ ▶ 1 of 2

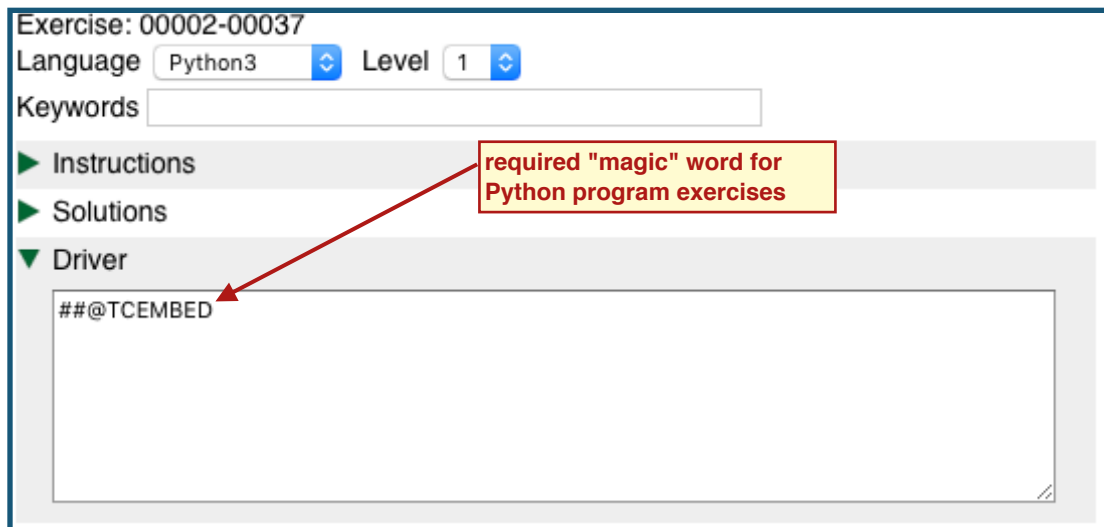
## SPECIAL ARRANGEMENTS FOR JAVA AND PYTHON

To the extent possible, the system treats all languages the same and provides as consistent an interface as possible. But languages do have differences, and in some cases have to be treated specially. Currently, there are minor differences for Java and Python.

**Java.** Because of the language-imposed file-naming requirements for source files, it is necessary for the exercise author to specify the *primary class*, that is the class whose static main method will be invoked:



**Python.** Because of the language-imposed formatting requirements (along with some internal system quirks), authors of Python Program exercises must also populate the DRIVER part of the interface as follows:



The magic word, "##@TCEMBED", will make slightly more sense after the discussion of "snippet exercises".

## CHANGING A MODEL PROGRAM EXERCISE

The reason for starting with a model program exercise, rather than providing a blank form, is to provide a clear reminder of what the nature and purpose of each form element is in the Exercise Editor interface. Once an author has cloned the model program exercise and has a clear idea of what exercise it is to be transformed into, the workflow is pretty much slash-and-burn: there is rather little that can re-used.

Let's suppose the exercise we wish to write involves processing a transaction file named "transactions" that consists of transactions, written one per line, each consisting of a one word transaction type (debit/credit) and an amount (an integer representing cents). The program is to create two files, named "debits" and "credits". The former gets the debit amounts in the order that they appeared in the "transactions" file and the latter gets the credit amounts in the order that they appeared in the "transactions" file.

Here is one set of actions to modify the clone of the model program exercise that was displayed above:

- (1) Open the Instructions and replace them with a clear, unambiguous statement of the new exercise.
- (2) Open the Solutions and replace the first and only solution with a working solution to the new exercise. (You may want to add additional solutions later-- typically you would click the **dup** button and modify your original solution rather than starting entirely from scratch.)
- (3) Open the Test Sets.
- (4) In the Test Sets, strip out the *command-line arguments*. (Leaving them would not hurt, since they are unused in this exercise, but their presence would be confusing on later inspection.)
- (5) In the Test Sets, open up *Input* and delete the first file by clicking the **Del** button.
- (6) In the Test Sets, with Input still opened, change the *file name* from "fileB" to "transactions". Then replace the *data* with appropriate test content.
- (7) In the Test Sets, close Input and open up *Output*. Change file name to "debits" and in the data area enter the data that a correct program would generate for the given transactions data.
- (8) Now click **New** and set the file name to "credits". In the data area enter the data that a correct program would generate for the given transactions data.

You've now replaced the instructions, the one solution, and the data in the one Test Set. Click on **Save** and observe the result.

Success is simple to display.

Failure is more complex.

One possible cause is a compiler error in the solution code— these are easily presented. See, for example, the diagnostics on the right, which are presented when a spurious space breaks up an identifier in a C program.

```
Tested on: macos
Your code did not compile:

./main.c: In function 'readSumAndWrite':
./main.c:5:20: error: expected '=', ',', ';', 'asm' or '__attribute__'
before 'tal'
    int n, result, to tal=0;
                      ^
./main.c:9:3: error: 'total' undeclared (first use in this function)
    total+=n;
    ^
./main.c:9:3: note: each undeclared identifier is reported only once for
each function it appears in
```

Another possible cause is a logical error in the solution code. *This is generally indistinguishable from an inconsistency in the Test Set specification.* The diagnostics on the right are presented when a mis-initialization has occurred in the solution code. Note that the message is of only minimal assistance: "Your output file ... is not what was expected". Turing's Craft is currently working to provide better diagnostics than this!

```
Tested on: macos
There was an execution error in your solution:

Execution Feedback: Checking output, Your output file fileZZZ is not what
was expected.

Solution:
//@ans
#include <stdio.h>

void readSumAndWrite(char *fname, FILE *fpout) {
    int    n, result, total=33;
    FILE *fpin = fopen(fname, "r");
```

## EXERCISE SPECIFICATION: SNIPPET EXERCISES

The interface for specifying **snippet exercises** is the same as that of **program exercises**. The Language, Level, Keywords, Instructions, Solutions, Initial Code and Options are all used in the same way. The difference is that *snippet exercises have a non-empty driver and don't necessarily depend on the Test Sets for playing a role in correctness determination.*

In the case of program exercises, the student provides an entire program, that is black-box tested, as per the tests specified in the Test Sets.

In the snippet exercises, the student provides a fragment of code that requires additional code to form a compilable and executable program. The additional code is called a **harness** and is placed in the Driver section of the exercise form. Let's consider the following Python3 example (which is what the model snippet exercise for Python3 is based on):

*Instructions:*

Assume that x is variable with an integer value. Write a statement that increases its value by 7.

*Solution:*

```
x = x+7
```

*Driver (FIRST DRAFT):*

```
import sys
x=55
##@TCEMBED
if x!=62:
    print("WRONG!",file=sys.stderr)
    sys.exit(1)
else:
    print("correct")
sys.exit(0)
```

Note the impossible-to-miss placeholder "**##@TCEMBED**". (In C, C++, and Java the placeholder would be "**//@TCEMBED**".) The placeholder indicates where the student's code will be inserted. If a student submitted

```
x = 7
```

then the resulting program would be:

```
import sys
x=55
x = 7
if x!=62:
    print("WRONG!",file=sys.stderr)
    sys.exit(1)
else:
    print("correct")
sys.exit(0)
```

When executed, the program would print the harsh and rather unhelpful message "WRONG!" to standard error and terminate with an exit code of 1. Had the student submitted

```
x = x + 7
```

or, more whimsically,

```
x = (x-1) + 8
```

The screenshot shows a web interface for specifying an exercise. At the top, there are two dropdown menus: 'Language' set to 'C' and 'Level' set to '1'. Below these is a text input field for 'Keywords'. A list of sections follows, each with a right-pointing triangle icon: 'Instructions', 'Solutions', 'Driver', 'Test Sets', 'Initial Code', and 'Options'. The 'Driver' section is currently selected and highlighted.

the program, when executed would print no message on standard error, would print "correct" on standard output, and would terminate with an exit code of 0.

Note that it is the *harness* (the instructor-provided code) that determines the correctness of the student code; it does so by:

- a) establishing an initial internal state (x has the value 55)
- b) examining the internal state to determine whether the student code modified the state in accordance with the specification of the instructions

In order to communicate to the MPL system that the student code is correct, the harness must do both of the following:

- a) terminate with an exit code of 0
- b) generate some standard output (it can be anything) that matches the specs of a Test Set.

In order to communicate to the MPL system that the student code is incorrect, the harness must do all of the following:

- a) terminate with a non-zero exit code
- b) fail to generate standard output that matches the specs of the Test Set
- c) generate some standard error message, which will be displayed as a hint to the student

The result of doing either of these partially is indeterminate.

In the simple example above, when the student code is correct, the harness produces the standard output "correct".

For that reason, there is a single Test Set, with no command-line arguments or input specified, and only one output target, namely standard output, whose expected data is the string "correct".

Some basic points about snippet exercises:

- \* they require at least one (typically only one) Test Set, with a short specification of standard output (and usually nothing else)
- \* submitted student code textually replaces the placeholder (e.g. "##@TCEMBED") wherever it appears to form a compilable and executable program (barring compilation errors in the student code)
- \* the harness determines correctness by examining the internal state of the program after the student code executes
- \* the harness communicates correctness to the system through a combination of exit status codes, standard output and standard error
- \* standard error messages are displayed to the student in the RESULTS tab as hints

A slightly better version of the above driver is:

```
import sys
x=55
##@TCEMBED
if x==55:
    print("You did not change the value of x.",file=sys.stderr)
    sys.exit(1)
if x!=62:
    print("the value of x was changed to an incorrect value-- it is not 7 more
than the original",file=sys.stderr)
    sys.exit(1)
else:
    print("correct")
sys.exit(0)
```

The advantage of this driver is that it conveys more information to the student than simply indicating "Wrong". Information from the internal state is gleaned to making reasonable guesses about the student code.

So, if the `x` is still equal to 55, perhaps the student had mistakenly written

```
x = x
```

or

```
c = x + 7
```

In both cases, the value of `x` was not changed and the hint will reflect that.

**Writing hints** that are guaranteed to be correct is just about impossible, a fact that is closely related to the work of this company's namesake. As an admittedly pathological example, the following student submission

```
x = x + 7
```

```
x = x - 7
```

certainly has changed the value of `x` over the course of execution but the analysis will still read "You did not change the value of `x`". (At least it won't say you did not assign anything to `x`!)

The full form for the Python Model Snippet exercise, the example we have just discussed is shown at the right.

The screenshot displays the Python Model Snippet exercise interface. It includes a toolbar with various icons for editing and formatting. The main content area is divided into several sections:

- Instructions:** A text box containing the instruction: "Assume that `x` is variable with an integer value. Write a statement that increases its value by 7." Below this is a text input field labeled "body p".
- Solutions:** A section with a "New" button and a "Dup" button. It contains a text input field with the solution: `x = x+7`. A status indicator shows "1 of 1".
- Driver:** A section containing a code editor with the following Python code:
 

```
import sys
x=55
##@TCEMBED
if x==55:
    print("You did not change the value of x.",file=sys.stderr)
    sys.exit(1)
if x!=62:
    print("the value of x was changed to an incorrect value-- it is not 7 more than the original",file=sys.stderr)
    sys.exit(1)
else:
    print("correct")
    sys.exit(0)
```
- Test Sets:** A section with a "New" button and a "Dup" button. It contains a "Command-line Arguments" field, an "Args" input field, and an "Input" field. Below these are "Output" options: "Stdout" (selected), "Stderr", and "File". A "Data" input field contains the text "correct". A status indicator shows "1 of 1".
- Options:** A section with checkboxes for "Ignore Whitespace", "To Single Whitespace", and "Ignore Case". Below these are radio buttons for "Plain Text: no regular expressions" (selected) and "RegEx: metachars are literal unless escaped".

