Teaching guide: Data validation and authentication

This resource will help with understanding robust and secure programming through data validation and authentication. It supports Section 3.2.12 of our current GCSE Computer Science specification (8520). The guide is designed to address the following learning aims:

- Recognise the inherent unreliability of data entered by a user.
- Investigate ways to validate data using code.
- Understand how file access can cause errors.
- Explore ways to validate usernames and passwords using an external data source.

A good rule of thumb when writing programs that involve user input is to always assume that the user will make mistakes and write your programs accordingly. By way of an example let's look at a guessing game, an instance of this game could look like this:

```
> Enter the number to be guessed
36
> Enter a guess between 1 and 100
fifty
$$$ Program crashed $$$
```

The user hasn't necessarily done anything wrong – they've entered the string `'fifty'` as their guess, because our computer program was expecting a string that could be converted to an integer, and because the string `'fifty'` cannot be directly converted, it caused the entire program to crash.

Different types of validation

In using a program, some users will make mistakes, and crashing the program every time they do will be frustrating. Code can be written that validates the user's input before carrying on. Validation doesn't mean that the user always enters something that is correct (such as a correct combination of username and password), but it means that the format of their user input is correct. A complete list of the necessary validation for our user's guess would be:

- minimum length (they need to enter something, this particular instance of checking minimum length is called a presence check)
- type (the input should be able to be converted to an integer)

- range (the integer representation of their input should be between 1 and 100).

The actions that result from the input not passing any of these validation attempts should be to prompt the user to enter another guess and, if we were being helpful, to be given a message that explains why their input wasn't valid. The high-level plan for this is:

1. user enters a guess
2. if the length of this guess is zero then
    a. output an error message
    b. go to 1
3. convert the input to an integer
4. if this causes an error
    a. 'catch' the error
    b. output an error message
    c. go to 1
5. if the integer is not between 1 and 100
    a. output an error message
    b. go to 1

At step 4a the error that would be caused by converting to an integer is 'caught'. This means that the program doesn't crash and when an error is found it is caught by a safety net around that part of code. We can use the following syntax in our pseudo-code to represent this safety net:

```
TRY
   # do some code that might cause an error
CATCH
   # if an error is found then do this code
ENDTRY
```

An example of `TRY-CATCH` when attempting to convert a string to an integer is:

```
string_input ← USERINPUT
TRY
   int_input ← STRING_TO_INT(string_input)
CATCH
   OUTPUT 'input could not be converted to integer'
ENDTRY
```

Coding validation routines

The entire validation procedure above should be written to continue to loop until the data is fully valid and, as we don't know how long this will be but it must happen at least once then a `REPEAT-UNTIL` loop would seem appropriate.

```
valid ← False
OUTPUT 'Enter a guess between 1 and 100'
```

```
REPEAT
   guess_as_string ← USERINPUT
   # presence check
   IF LEN(guess_as_string) = 0 THEN
      OUTPUT 'You have not entered anything'
   ELSE
      TRY
         # type check (will go straight to CATCH if it fails)
         guess ← STRING_TO_INT(guess_as_string)
         # range check
         IF guess < 1 OR guess > 100 THEN
            OUTPUT 'Must be between 1 and 100'
         ELSE
            # if all checks passed then input is valid
            valid ← True
         ENDIF
      CATCH
         OUTPUT 'Must enter an integer (e.g. 42)'
      ENDTRY
   ENDIF
UNTIL valid
```

File access often causes errors if the file does not exist. We can make programs more robust (ie better able to handle unexpected events) by accounting for this in our program.

Using an example of reading a file of scores we can check the validity of the filename:

```
TRY
   file ← OPEN('scores.txt')
   data_in ← READLINE(file)
   CLOSE(file)
   # carry on processing the data from data_in
CATCH
   OUTPUT 'File does not exist'
ENDTRY
```

If we wanted the user to enter the filename themselves and, should the file not exist, be prompted to enter another filename then the code could be easily extended:

```
valid ← False
REPEAT
   OUTPUT 'Enter a filename'
   filename ← USERINPUT
   TRY
      file ← OPEN(filename)
      data_in ← READLINE(file)
```

```
        CLOSE(file)
        valid ← True
    CATCH
        OUTPUT 'File does not exist'
    ENDTRY
UNTIL valid
# carry on processing the data from data_in
```

The code below can be used as a way to get user input between a lower and upper bound:

```
SUBROUTINE get_input(lower, upper)
    OUTPUT 'Enter a number between'
    OUTPUT lower
    OUTPUT 'and'
    OUTPUT upper
    number ← USERINPUT
    RETURN number
ENDSUBROUTINE
```

We can combine our knowledge of validation and subroutines now to amend this code so the subroutine will not return a value unless it is an integer within the correct range (this example uses a `WHILE` loop instead of `REPEAT-UNTIL` for variance):

```
SUBROUTINE get_input(lower, upper)
    OUTPUT 'Enter a number between ' + lower + ' and ' + upper
    number_as_string ← USERINPUT
    # continue to loop until number is returned
    WHILE True
        TRY
            number ← STRING_TO_INT(number_as_string)
            IF number < lower OR number > upper THEN
                OUTPUT 'Number not within bounds'
            ELSE
                RETURN number
            ENDIF
        CATCH
            OUTPUT 'Not an integer'
        ENDTRY
    ENDWHILE
ENDSUBROUTINE
```

Data validation code is often very repetitive, so becoming adept at putting this code inside subroutines that can be reused will speed up the development of the code as well as making the code more structured, shorter and less likely to contain errors.

Authentication routines

Both usernames and their associated passwords are stored persistently between instances of a program, using the techniques so far we could store a username followed by that user's password on one line of a text file. If we wanted to check if a user has entered a correct username/password combination then we could loop over every username-password in this text file until there is a match and, if so, let the user progress. As an overview:

1. User enters a username
2. User enters a password
3. Open the file with usernames and passwords
4. Loop over every username-password combination in the file
    a. if the usernames match then check the passwords match
    b. if the passwords also match then allow the user to progress

There are many finer details that would be involved in implementing this such as validating the user input to make sure they have entered a username and password, dealing robustly with file access, closing file connections and ensuring that the loop terminates either when no further usernames are available or when a successful username-password match has been found.

Furthermore, storing usernames and passwords as unencrypted text is insecure and would not be suitable for a real-world application, however implementing encryption goes beyond the requirements of the specification.