Programming Merit Badge Guide

Arduino

Welcome to the world of programming! We will start our Programming merit badge by modifying a simple C-based program for “Arduino”. The Arduino is a prototyping platform for the Atmega328 chip (a prototyping platform is a hardware design which offers easy access to a given chip, without the need to design circuit boards or solder wires). The Atmega328 chip is used universally around the world, in everything from automobile braking systems to refridgerators and thermostats. And you’re about to take control of one!

Sketches

Arduino code is written in what is referred to as a “sketch”. A sketch has three important parts:

1. References, constants and global variables, which come at the head of the sketch.
2. The setup() function, which sets up the program and prepares it to function.
3. The loop() function, where the logic all takes place

For our Calculator sketch, we don’t need to pay much attention to anything other than the loop() function, but it is interesting to look at the setup() function and see that this is where we print out instructions to the user, with the Serial.println() method.

User Interaction

The Arduino was NOT written as a user-interactive device, so our application is a bit kludgey. All our input is via the command line, and it’s not exactly logical how it works. Just remember:

- To add, enter two numbers separated by commas, and then the ‘+’ symbol: 4,5+
- To subtract, enter two numbers separated by commas, and then the ‘-’ symbol: 4,2-
- To multiple, enter two numbers separated by commas, and then the ‘x’ symbol: 4,5x
- To divide, enter two numbers separated by commas, and then the ‘/’ symbol: 20,5/

We could have purchased touchscreens for each Arduino, and drawn a calculator UI. That would have made things very convenient for the user but would have cost about a hundred dollars.

The Calculator Function

Our application is a calculator. It performs math operations against numbers, but the Arduino user-input is string based. So if you scroll to line 56 where the math functions start, you'll see that our first step is to create some variables. Specifically:

```
int answer = 0;
int y = int(values[0]);
int z = int(values[1]);
```

What's going on here?

1. First, we create an integer variable (a variable is a place in memory where something will be stored) and set it to 0.
2. Next, we create an integer called 'y' and set it equal to the user input - but we do something special, and turn the user input into an integer by using the int() function. This is called "casting" (we 'cast' a string to an integer - and hope the user actually inputs numbers)
3. Finally, we create another integer, this one called 'z', for the second number provided by the user.

Doing Math

Now that we have numbers, it's time to figure out which mathematical operation the user wants to perform, and then go do the work. We do this by implementing a switch statement - this is program flow logic which is based on multiple possible values for a given statement. In our case, we are "switching" on 'readChar) which is the final character provided by the user.

- If the character is a '+', we will do addition.
- If the character is a '-', we will do subtraction.
- If the character is an 'x', we will do multiplication (note: the asterisk is a reserved character in Arduino, so we have to use x instead of * to indicate multiplication).
- If the character is a '/', we will do division.

In each case in the switch statement, we print out some information to the user:

1. Text indicating the start of the answer:
   `Serial.print("The product of ");`
2. The two values provided by the user,
   `Serial.print(values[0]);` and
   `Serial.print(values[1]);`
3. and then the answer itself, which is calculated by using a helper function:
   `answer = doDivision(y, z);`

If you skip down to the helper functions, you'll see they are quite readable and make sense - they are logical (of course, since computers simply run on logic). Scroll to line 113 where the doAddition helper function is found.

This function starts with a declaration:

```
int doAddition(int a, int b)
```

The 'int' at the beginning of the function declaration indicates the function will return an integer. Next comes the function name (doAddition) then come the function's parameters, which is input the function expects, and must have in order to process.

Inside the function we see what looks like simple algebra. It's definitely simple:

```
int sum = a + b;
```

Finally, there is a return statement, instructing the program to return the answer:
return sum;

Oh and those curly braces… All they do is help make the program more readable through organization. You're about to embark on an epic battle: where do curly braces go? In C# (.NET), they are often on the line after the declaration. In Java, they're on the same line. In C, it varies… It's a matter of personal preference. In our Arduino code, the braces are on a new line and in our Java code they are on the same line. Which do you prefer?

Write The Next Function

The doAddition function has been written for you. Can you follow the same pattern, and implement functions for:

- doSubtraction
- doMultiplication
- doDivision

Sure you can - it's simple!

doSubtraction()

For the doSubtraction function, follow the logic for doAddition, but change a few things:
int doSubtraction(int a, int b)

```
{
  int difference = a - b;
  return difference;
}
```

You'll see we named the variable "difference" rather than sum, so our return statement is a bit different as well.

doMultiplication()

You try this one, naming the answer “product”. Don’t forget that, in C, the symbol for multiplication is an asterisk (‘*’).

doDivision()

And I’m sure you can do this one, too. Remember the output of a division operation is the quotient. The symbol for this operation is a forward slash: ‘/’.

Your math code should ultimately look like this:

```
int doAddition(int a, int b)
{
  int sum = a + b;
  return sum;
}

int doSubtraction(int a, int b)
{
  int difference = a - b;
  return difference;
}

int doMultiplication(int a, int b)
{
  int product = a * b;
  return product;
}

int doDivision(int a, int b)
{
  int quotient = a / b;
  return quotient;
}
```

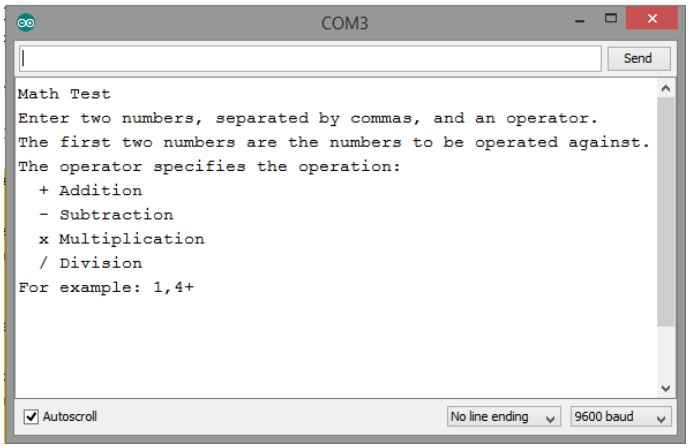
Try It Out!

Now that your code is written, it’s time to run it. Click the “Upload” icon in the Arduino IDE.

It looks like an arrow pointing right: 

This compiles your program and pushes it onto the Atmega chip.

Next, click the “Serial Monitor” option on the Tools menu, which opens the Arduino serial monitor. It will look like this:



Start testing out your work. Enter a series of addition, subtraction, multiplication and division problems and see how they work. As you test, start to do crazy things like adding really large numbers (try adding 67000 and 67000). You’ll notice something strange:

- For really large answers, the answer you get is very strange. This is because the largest integer on Arduino is about 32,000. When the answer exceeds that amount, the sum ‘wraps’ around and starts adding again from 0. It’s like a clock… 7 o’clock plus 7 hours is 2 o’clock; the value ‘wraps around’ 12 and starts over at 1.
- For division that results in a decimal, your answer is always rounded up. That’s because an integer is only a whole number, and computers don’t round down. So 7 divided by 2 will be 3.

We’ll learn about more accurate number types later during the Android project.

Python

Python is an interpreted language and a hybrid of spoken languages and programming language. Python is a truly powerful language, but quick to learn with plenty of support libraries. It is the basis for many web sites and is used extensively in the IT industry to manage servers.

Code Structure

Python code is written and stored in text files, typically with a .py extension. Python is an object-oriented language. To run Python natively on your machine requires an extensive installation, but you can run Python code in a web browser at www.codeskulptor.org, which is how we’ll work on the calculator project.

Lesson: Implement Math

Continuing with our calculator theme, we will be working with a Python-based calculator. This calculator works different from most calculators you buy at a store: it is an engineering-oriented calculator which allows you to place numbers on a stack. So rather than key-clicking “1” “+” “5” “=”, you will enter “1” “Enter” “6” Enter” “+” “Enter”.

Our code looks a lot different from the C code we just used for Arduino. The first thing you may notice is that the variables are not typed. In Arduino, to declare a variable we gave it a type, a name, and often a value: `int answer = 0;`

In Python, we just give it a name and a value: `stack1 = 0`

Also, you probably noticed there isn’t as much “punctuation” (no semi-colons to end a statement). For folks just getting started, this can simplify the coding experience. For engineers migrating from C to Python, well, it can be annoying!

Our code begins with a series of declarations. These are variables we’ll be using for the project. There are a series of logic functions starting with “`output()`”, which prints the answer to the screen. The `swap()` function allows you to change the order of the input.

For which math operation does order matter?

Next is the “`state()`” function, which tracks the state of the calculator to determine whether the first number has been entered by the user. And finally, we come to our math functions!

First: Addition

The addition function looks a lot different from our Arduino code:

```
def add():
    """ add operand to store"""
    global stack1, stack2, temp_num, calc_start
    state()
    if calc_start == "Started":
        stack1 = stack1 + temp_num
    elif calc_start == "RPN":
        stack1 = stack2 + stack1
        stack2 = 0
        calc_start = "Started"
    output()
```

Let’s break this down…

The “global” line references global variables available throughout the program. If these variables weren’t declared as global, they would only “exist” in the context of the function and couldn’t be used elsewhere - for instance, they could not be used in “`output()`” to print the answer. The next line calls “`state()`” to determine what state the application is in - whether it has the first number or not.

There are two similar math functions, based on a conditional: if the state is `started` (meaning the user has already entered a number, but only one) take the number entered and add to it the number in the text field in the user interface. If the state is “`RPN`” we know we have both numbers and so we operate on stack1 and stack2, our two variables. Finally, we call `output()` to display the results.

Notice we have an interesting `if` statement:

```
if calc_start == “Started”;
  do something
elif calc_start = “RPN”
  do something else
```

Two things of note:

1. There is an if, and then an “else if”. The logic here is exclusive, meaning if the first or second conditions are met, do something. Otherwise, do nothing.
2. elif is a Python abbreviation for elseif. Welcome to Python, and you have an inkling of an idea why developers with a C background go a little crazy when they first start coding in Python.

Second: Subtraction

Now you are going to write the subtraction function. I bet you can do it quickly - it’s a copy and paste of the addition, but you have to change the operator. So it will look like this:

```
def sub():
    """ subtract operand from store"""
    global stack1, stack2, temp_num, calc_start
    state()
    if calc_start == "Started":
        stack1 = stack1 - temp_num
    elif calc_start == "RPN":
        stack1 = stack2 - stack1
        stack2 = 0
        calc_start = "Started"
    output()
```

Third and Fourth: Multiplication and Division

Can you do the multiplication and division yourself? Sure you can - give it a try, it should be easy to copy and paste and then modify one or your other functions.

Now press the “Play” button and see if your application works!

Android

Android Source Organization

We won’t take too much time, but here are the main components of Android source (found in the “app/src/main” directory):

- Java: your source code is here. For this project, we have twoJava files (referred to as a ‘class’ or a ‘class file’):
    - main_activity.java: the application logic and UI can be found here
    - k7jtoMath.java: the math logic is all contained here
- Resources
    - drawable-xxxxx: these are icons for your application
    - layout: contains xml files which describe the UI for your application. You can hand-modify these files, or view them in design (preview) mode.
    - menu: xml files which define various menus in your application. We only have menu_main.xml, which shows a basic about box.
    - values: the important file here is “strings.xml” where all the UI text is stored.

For this tutorial, most of our work will be in the k7jtoMath.java file.

Classes

A class is a self-contained object of code which can provide functionality (like doAddition) but also other objects like variables. Classes are great because they can be reused--an engineer can build one class and use it in several programs. Classes are encapsulated, meaning their functionality is self-contained and is pretty much a black box to the rest of the appliction. The k7jtoMath class is a good example of this - the main_activity.java class calls the k7jtoMath class to perform certain functions. The main_activity class doesn’t care how k7jtoMath works, it just wants a string back so it can display the string to the user.

Java is full of classes - a well-designed enterprise Java application can have scores, if not hundreds, of classes.

Lesson: Implement Math

Open the k7jtoMath.java file in Android Studio. Scroll down to around line 22 where you see the comment “Add math functions here”. Before we can improve, we actually have to write the math code!

Just like with the Arduino and the Ruby calculators, there are four functions to implement:

- doAddition
- doSubtraction
- doMultiplication
- doDivision

Can you guess how to implement them? Sure you can - they're almost like the C you implemented in Arduino, so this will feel very familiar.

First: Cast String to Int

For simplicity's sake for the refactoring exercise, the math functions have been implemented to accept and return strings. You would never do this in a real math function! The first step for each math function is to caste the incoming strings into integers.

Casting a number is a relatively simple function. If you scroll up just a few lines to "getNumber()" you will see the following code implemented:

```
    public int getNumber(String numIn) {
       try {
           return Integer.parseInt(numIn);
       }
       catch (Exception ex) {
           // TODO: need to bubble up theexception
           return 0;
       }
   }
```

There's nothing magical about this code. The "`try`" statement tells the computer to tread carefully and, if there's an error, not to panic but to turn the error over to the application. The next line, "`return Integer.parseInt(numIn);` does three things:

1. The `return` command instructs the application to return the value from the rest of the line.
2. The `Integer.parseInt` command instructs the application to read the literal string and try to convert it to a number. This would succeed for a string like "100" whereas it would fail for a string like "seven".
3. The next line, `catch`, is where any errors from the try statement are handled. I've been lazy and implemented a very basic catch command.

Second: Implement doAddition

It's time to write the doAddition function, which starts by casting strings to ints:

These two lines call the “getNumber” function, which casts the string to an integer (you’ll be changing that later in the exercise).

```
int numX = getNumber(a);
int numY = getNumber(b);
```

Now that your strings have been cast to ints, do the math:

```
int numAnswer = numX + numY;
```

Finally: Return Answer

Lastly, return your answer (in the form of a string):

```
return Integer.toString(numAnswer);
```

Pretty simple, right? OK - you try the other three. Do the same 4 lines for doSubtraction, doMultiplication and doDivision (but don’t forget to change the operator).

Final Code

Your code for doAddition, doSubtraction, doMultiplication and doDivision should look like this:

```
 public String doAddition(String a, String b) {
    int numX = getNumber(a);
    int numY = getNumber(b);
    int numAnswer = numX + numY;
    return Integer.toString(numAnswer);
}

public String doSubtraction(String a, String b) {
    int numX = getNumber(a);
    int numY = getNumber(b);
    int numAnswer = numX - numY;
    return Integer.toString(numAnswer);
}

public String doMultiplication(String a, String b) {
    int numX = getNumber(a);
    int numY = getNumber(b);
    int numAnswer = numX * numY;
    return Integer.toString(numAnswer);
}

public String doDivision(String a, String b) {
    int numX = getNumber(a);
    int numY = getNumber(b);
    int numAnswer = numX / numY;
```

```
        return Integer.toString(numAnswer);
    }
```

Test Your Code

No decent programmer releases their code without validating it. Run some test cases against your code.

- Run some valid cases for each function - addition, subtraction, multiplication and division.
- Next, try some unexpected number combinations. Pay special attention to using really, really big numbers (like multiplying huge numbers by themselves). Multiple 456*456*456*456 several times. What happens?
- Try division problems which result in decimal numbers. What happens?
- Try to divide by zero.

You should have noticed that division which ends in decimal numbers (non-whole numbers) has funky results - the answer is wrong, it's rounded to a whole number. That's because integers cannot hold decimals, so the processor automatically rounds up. That's fine if you're dividing oranges among friends, but if you're making astrophysics calculations, that kind of precision just won't do! It's time to move to the next lesson: decimal vs. integer.

Lesson Goal: Decimal vs Integer

Two definitions:

- Integer: an integer is a whole number. 5, 102, 190847 are all integers. When you were using the Arduino you probably discovered that integers can't exceed 32,000. On the Android platform, integers can't exceed 2147483647 (positive or negative).
- Decimals are non-whole numbers. 1.29, 3.14, and .000379291 are all decimals. In Java, decimals are called "doubles"

Why does this matter? In the early days of computing, available memory was measured in bytes (today we measure it in gigabytes, or trillions of bytes). Programmers had to be very careful with memory. Integers were the simplest numbers - they took up the least amount of memory. So if a whole number less than 32,000 would do, a developer used an int.

Decimals and other number types use more memory but offer greater precision. For this lesson, we are going to convert all of our ints to decimals. Let's get started!

Refactoring: Step With Care

It's a good idea to make one change at a time, to "compile early, compile often". This allows you to catch errors when they are introduced, rather than having to unravel a long string of changes to find where the error was introduced.

Refactor getNumber()

Remember the `getNumber()` function is where all of our strings are cast to numbers. This is the best place to start.

1. Change the return type from "`int`" to "`double`"
2. Change the cast statement from `Integer.parseInt(numIn)` to `Double.parseDouble(numIn)`

That's it! What happens when you try to compile, though? Yes - lots of errors, because throughout your math you are expecting getNumber() to return an integer, and now you're returning a double.

Refactor the Math Functions

Next, you'll refactor all 4 math functions. We'll demonstrate `doAddition()` here, and you're on your own to do the others.

In `doAddition()`, make the following changes:

1. For each "`int`" statement (like '`int numX`') change the data type from `int` to `double`:
   `double numX = getNumber(a);`
2. Next, change the data type for the `int numAnswer` statement:
   `double numAnswer = numX + numY;`
3. Finally, change the `return` function to return a string parsed from a double:
   `return Double.toString(numAnswer);`
4. Lastly, build your project and make sure no errors remain in doAddition.

Simple enough right? Now you try - refactor the other three. Don't forget to build after each change. Step by step, your errors should become fewer and fewer until the build compiles successfully again. At this point, your math should look something like this:

```
 private double getNumber(String numIn) {
        try {
            return Double.parseDouble(numIn);
        }
        catch (Exception ex) {
            // TODO: this is horrible, need to bubble up the exception
            return 0;
        }
    }

//*****************************************************************
//                Add math functions here
//*****************************************************************
```

```
public String doAddition(String a, String b) {
    doublenumX = getNumber(a);
    double numY = getNumber(b);
    double numAnswer = numX + numY;
    return Double.toString(numAnswer);
}

public String doSubtraction(String a, String b) {
    double numX = getNumber(a);
    double numY = getNumber(b);
    double numAnswer = numX - numY;
    return Double.toString(numAnswer);
}

public String doMultiplication(String a, String b) {
    double numX = getNumber(a);
    double numY = getNumber(b);
    double numAnswer = numX * numY;
    return Double.toString(numAnswer);
}

public String doDivision(String a, String b) {
    double numX = getNumber(a);
    double numY = getNumber(b);
    double numAnswer = numX / numY;
    return Double.toString(numAnswer);
}
```

Now run the same cases you ran before, paying close attention to division as well as operations which result in really large numbers.

Lesson Goal: switch statements (semi-advanced logic)

Do you remember talking about John van Neuman? He developed the concepts which led to object-oriented programming. One of his greatest contributions was logic, or “control flow”. When we think of control flow, we naturally think of “if” statements, like this:

```
if (something is true) then {
  do something
}
else {
  do something else
}
```

Well, van Neuman's control flow was great, until you tried to apply it to multi-state situations like our calculator. Take a look at the code in main_activity.java, which starts on line 68. This is the function appendNumber().

Remember computers do exactly what you tell them to do, and only what you tell them to do. Our number pad consists of several buttons which, when tapped, place a new character on the display line (appended to the end of the current string). There are 10 number buttons--I could have written a function for each number button, like this:

```
public void appendNumberOne() {
  // some UI-based logic to get to the edit text here
  myEditText.setText(myEditText.getTest() + "1";
}

public void appendNumberTwo() {
  // some UI-based logic to get to the edit text here
  myEditText.setText(myEditText.getTest() + "2";
}

public void appendNumberThree() {
  // some UI-based logic to get to the edit text here
  myEditText.setText(myEditText.getTest() + "3";
}
```

But if I did it this way, I'd have 10 methods which do essentially the same thing. And if there were a bug in my logic, I'd have to maintain it in 10 places. If there's one thing about developers, it's that we are lazy. Why do something 10 times, when we can do it just once? If there were a way to write one function which detects which button has been tapped, my work would be reduced by 90%.

One solution would be to use a series of if statements, but that would be cumbersome, since there would be 10 if statments (which means 20 curly braces, which means a high likelihood for error):

```
if (buttonTapped = buttonOne) {
  myEditText.setText(myEditText.getTest() + "1";
}

if (buttonTapped = buttonOne) {
  myEditText.setText(myEditText.getTest() + "1";
}

if (buttonTapped = buttonOne) {
```

```
        myEditText.setText(myEditText.getTest() + "1";
    }
```

…

So now enters the “switch” statement. This allows you to ‘switch’ on a given test, and reduces the program flow overhead. Check out the switch statement on line 76:

```
    switch (v.getId())
```

This instructs the application to do a variety of things, based on the valid of the ID of the button which has been clicked. In line 78, if button1 has been clicked the program appends the character “1” to the end of the editText. In line 90, the program appends the character “5” if btn5 has been clicked, and so on.