CHAPTER 0 – LEARNING

Preliminaries

LEARNING . . . LEARNING . . . LEARNING

L0.1 – COMPUTER SCIENCE (CS)

L0.1.a – A Broad Definition

Everybody agrees that CS has played a pivotal role in shaping human history. Almost every advance in the past few decades in science, engineering, medicine, and technology has CS to thank for it. And besides being a framework and catalyst for other fields, CS can directly help us understand how Nature itself works because CS is about information and increasingly, the basic building blocks of Nature appear to be information based[†].

As a relatively young discipline, CS does not have a universally adopted definition. And due to its ubiquitous nature, you will get different definitions depending on the research area of the person you ask. The author's take on this draws on the definitions of older sciences:

- *Physics is about matter and forces and it studies their interaction.*
- *Biology is about living things and it studies their evolution.*

We posit:

- *Computer Science is about information and it studies its transformation.*

As a simple example, when you store a piece of information, such as a number or a name, in your phone, the CS in the phone must transform the sequence of key presses that you make to a form that can be represented in memory. Performing a computation, compressing a picture, or encrypting a file does not *create* information; it merely *transforms* it.

By studying information, CS paves the way to answering such questions as "can anything be transformed or are there limits to computing?". The *CS Trail* sections of this book provide a glimpse of the answers at a level appropriate for this book. Albeit informal, it brings to focus some of the foundational concepts of CS and piques the reader's interest.

Unlike other sciences, which describe transformations through continuous math (formulae, calculus, differential equations, etc.), CS describes them via discrete processes. And although the two formalisms are related at a deep level, the CS description promotes *process-based* (also known as *computational*) thinking, which is a hallmark of CS.

We will refer to the process of transforming information as an *algorithm*. The algorithm does not concern itself with how information is represented, only with how it is transformed. An algorithm together with the description of the information is known as a *program*.

[†] In this view, particles, fields, living cells, and perhaps spacetime itself may all be emergent from information.

L0.1.b – Programming

We found that a program consists of a description of information (such as two integers of a given range) and an algorithm to transform the information (such as finding their sum). The task of *programming* thus involves finding a data structure to represent the information and devising an algorithm to carry out the transformation.

But even after the program is designed, we still need to cast it in a form that can be executed by a computer, and this is no small feat because computers can only execute a very small set of operations. These operations, collectively known as the elements of *machine language*, are so primitive that it would be extremely protracted and unwieldy to use them directly to write programs. CS confronts this challenge through a sequence of intermediate translators that allow us to write programs in a high-level language. For example, when you write a program in *Java*, the first layer of translators will turn your program to an equivalent one written in a lower-level language known as *bytecode*. Next, the second layer of translators, known as *just-in-time compilers*, will translate bytecode into machine language plus a few *system calls*. Next, a third translation layer within the operating system (the *linker*) replaces the system calls with machine code, so the program would finally be expressed in the language of the machine.

Using intermediate layers like this is a recurring theme in CS, and although it pops up under different names such as indirection, abstraction, and separation of concerns, the essence of it is always the same. We will encounter this theme repeatedly in this book so it is important that you take some time to think it through and be comfortable with how it works. To that end, here is a second example: to make something bold in HTML, you would write:

```
<b>some text</b>
```

The `<b>` tag is not a machine language instruction, so what happens here is that the browser will replace it with several instructions to be sent to the rendering engine, which in turn, will replace them with other instructions that will ultimately be replaced with machine language.

There are many *general-purpose* high-level programming languages in use today (C, C++, Java, Python, JavaScript to mention a few) and each has its advantages and disadvantages. (There is no best or worst language but there are bad and good programmers!). In addition, there are many *special-purpose* languages that are suited for specific applications or environments, such as HTML, XML, CSS, SQL, and MATLAB to mention a few.

We will be using **Java** in this textbook to write Android apps and **XML** to represent the user interface and the various resources of our apps.

L0.2 – A BRIEF ENCOUNTER WITH JAVA

L0.2.a – Creating a Java Class

Unlike the *Doing* parts (for which we create one project per chapter), we will create just one project for the *Learning* parts of all chapters. Hence, create a new project following the same steps as in Section D0.2.a except for two things:

- Select *No Activity* instead of *Empty Activity* for the default screen.
- Name the project *TestBed*.

Since all our Java classes will reside in this project, we will arrange them in packages, one per chapter, to better organize and to avoid naming conflicts. All these packages will be placed under the *main package* as sub-packages. Follow these steps to create them:

1. Right-click the main package; i.e. *not* the test packages.
2. Select *New* and then *Package*.
3. When prompted, enter ***zero*** for the package name. Since this is a sub-package of the main package, its name will dot-append the name of the main package.
4. Repeat the above to create five more sub-packages named ***one***, ***two***, ***three***, ***four***, and ***five***. You can also add a ***scratch*** sub-package for your own scratch work.

The smallest executable building block in Java is the **class**. Classes in Java can be designed in two different paradigms: *functional* or *object-oriented*. We will use the former in this chapter and discuss the latter in the next. Follow these steps to create your very first class:

1. Right-click the *zero* sub-package.
2. Select *New* and then *Java Class*.
3. Enter `Rectangle` for the class name.

The Project Pane should now show a new class in this package, and the Editor Pane should now have a new tab named `Rectangle` open, and it contains:

- A `package` statement at the top.
- The class header.
- An open and close brace between which the *class body* will be sandwiched.

Note that the placement of the braces is a matter of style. One approach is to place the open brace on a line by itself after the header line (as shown below on the left). Another is to place the open brace on the same line as the header (as shown below on the right):

```
public class Rectangle          |    public class Rectangle {
{                               |       ... (class body)
   ... (class body)             |    }
}                               |
```

Both placements are in common use and it is up to you to use one or the other, and you can configure the IDE to make either the default. We adopt the former style (shown on the left) when we write code fragments in this book because its open and close braces are vertically aligned, and hence, it makes it easy to spot a missing brace. Note that a brace "{" is different from a bracket "[" and different from a parenthesis "(".

Complete the development of the class as shown below:

```
public class Rectangle
{
    public static int getArea(int width, int height)
    {
        int result = width * height;
        return result;
    }
}
```

It is instructive to type these lines manually and watch how the IDE behaves as you type. If you make a typing mistake, the IDE will detect the problem as you type, and in that case, it will highlight the mistake in red and suggest a fix.

You are not expected to fully understand the code at this stage but take some time to observe and think through the following points:

- The class body has a *method* named `getArea`. It too has a header and a body sandwiched between two braces.
- The method header has the word `static`. This is a hallmark of the functional paradigm: all methods in it must be `static`.
- The method header indicates that it takes two integers named `width` and `height` and that it returns an `int`. The return type appears after the word `static`.
- It thus acts as a function or a service: you give it two integers; it works on them and then returns some integer.
- The method header has the word `public`. Its service can thus be used by other classes.
- The method body computes the result by multiplying the two given integers. It returns their product to the class that requested the service.

It is evident from the names of the class and the method, and from the computation in the method body, that it is intended to compute the area of a given rectangle. To increase our comfort level with classes, let us augment the class with a second method that computes the perimeter of a given rectangle.

Here is the thought process step by step:

- To make the method name indicative of what it does, let us call it **getPerimeter**. Making the method name a verb is best practice.
- The method header should indicate that it is **public** so we can use it from other classes.
- It should also indicate that it is **static**, per the functional paradigm.
- To compute the perimeter, we need to know the width and height of the rectangle, so our method should two take integer parameters.
- If the two sides of the rectangle are integers then its perimeter should also be an integer. Hence, the method's return type should be **int**.

This leads us to the method header:

```
public static int getPerimeter(int width, int height)
```

We now switch from *what* to *how*. Given *what* the method does, *how* do we implement it? The perimeter is defined as the sum of all four sides; i.e. twice the width plus twice the height, or, twice the sum of the width and height.

The complete class now looks like this:

```
public class Rectangle
{
    public static int getArea(int width, int height)
    {
        int result = width * height;
        return result;
    }

    public static int getPerimeter(int width, int height)
    {
        int result = 2 * (width + height);
        return result;
    }
}
```

L0.2.b – Testing a Java Class

To determine if a class meets its requirement, we must test it. This can be done in a number of ways, chief among them is the so-called JUnit testing[†]. We will present it in a simplified way in this chapter and revisit testing in general in the next chapter. Follow these steps to set up testing sub-packages, one per chapter:

1. Right-click the *Test* package; i.e. *not* the main and *not* the androidTest package.
2. Select *New* and then *Package*.
3. When prompted, enter ***zero*** for the package name. Since this is a sub-package of the Test package, its name will dot-append the name of the Test package.
4. Repeat the above to create five more sub-packages named ***one***, ***two***, ***three***, ***four***, and ***five***. Add also a ***scratch*** sub-package if you added one under the main package.

We have established a correspondence between the main and the Test packages by creating sub-packages with the same name in both. To test a class `C` in any main sub-package, we will create a class named `CTest` in the corresponding Test sub-package. Let us test the `Rectangle` class created in L0.2.a. It resides in the *zero* sub-package of the *main* package:

1. Right-click the *zero* sub-package of the *Test* package.
2. Select *New* and then *Java Class* and enter `RectangleTest` for the class name.

Type in the class as follows:

```
public class RectangleTest
{
    @Test
    public void getAreaTest()
    {
        int w, h;
        System.out.println("Testing getArea:");
        w = 4; h = 3;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getArea(w, h));
    }
}
```

When you use a feature (such as the above `@Test` annotation) that needs to be imported, the IDE will flag it in red and ask you to press `Alt-Enter`.

[†] Testing can also be done through a `main` method but we will stick to JUnit in this book.

As you can see, the testing class contains a method named `getAreaTest` and it is intended to test the `getArea` method in `Rectangle`. The body of the method picks a particular test case (a `4` by `3` rectangle) and sends it to the method to be tested. It outputs the test case that was used and the return of the method. To run the test, right-click the class header and select *Run*. You should see the output of the test in the Run pane at the bottom:

```
Testing getArea:
4,3
12
```

This indicates that the `getArea` method works as expected for this test case, because `4 x 3` is indeed `12`. Let us test a second case, say a `2` by `7` rectangle:

```
public class RectangleTest
{
    @Test
    public void getAreaTest()
    {
        int w, h;
        System.out.println("Testing getArea:");
        //====================================
        w = 4; h = 3;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getArea(w, h));
        System.out.println("--------------------");
        w = 2; h = 7;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getArea(w, h));
    }
}
```

Re-run the test and the following output should appear:

```
Testing getArea:
4,3
12
--------------------
2,7
14
```

Summary: To test a `static` method `m` in class `C`, create class `CTest` (under the Test package) and put in its body an `@Test` method with the header: `public void mTest()`. In the method body, and for each test case x, invoke the method `m` using `C.m(x)` and output its return.

As a practice of this general process, add a test for the perimeter method and re-run:

```
public class RectangleTest
{
    @Test
    public void getAreaTest()
    {
        int w, h;
        System.out.println("Testing getArea:");
        //===================================
        w = 4; h = 3;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getArea(w, h));
        System.out.println("--------------------");
        w = 2; h = 7;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getArea(w, h));
    }

    @Test
    public void getPerimeterTest()
    {
        int w, h;
        System.out.println("Testing getPerimeter:");
        //=======================================
        w = 4; h = 3;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getPerimeter(w, h));
        System.out.println("--------------------");
        w = 2; h = 7;
        System.out.println(w + "," + h);
        System.out.println(Rectangle.getPerimeter(w, h));
    }
```

L0.2.c – Installing and Testing a Java Library

The standard Java library is bundled with the IDE so you don't need to install it in order to use it. The same applies to the Android platform library. For other, third-party libraries, however, you must first fetch the library and store it on your development machine, and then incorporate it (as a dependency) into your app.

Most Java libraries come in the form of a **JAR** (Java Archive) file. This is just one file that contains all the classes of the library. After downloading this file from its maker, you must store it the subfolder named **app/libs** under the app's folder in `AndroidStudioProjects` and then link it with your project.

Let us install the **i2c** library associated with this book:

1. Download the **i2c.jar** file from **http://book.roumani.ca/**
2. Do *not* open or unzip the file.
3. Store the downloaded file in **AndroidStudioProjects/TestBed/app/libs**
4. Click *Project Structure* in the *File* menu.
5. Click *Dependencies* in the left pane and then *app* in the middle pane.
6. Click + at the top of the right pane then *Jar Dependency*.
7. Click the `libs/i2c.jar` file (in the drop-down list) or type it in.
8. Keep the default *implementation* configuration then click OK

The first three steps *install* the library while the last five *link* it to your project. This way, when it is time to translate your code to machine language, the translator (aka **gradle**) will look inside the jar file to complete the build process.

The `i2c` library was designed as a pedagogic scaffolding tool. You will need to add it to all the *Doing* projects that you will build in this book. Whenever you create a new project, copy the jar file from the above location to the `app/libs` subfolder of the new project then link it.

To verify that the library has been correctly installed and linked, let us use a feature in it and verify that it responds as expected. The feature we will use is the `static` method `repeat` in class `Utility`. If you invoke it like this:

```
Utility.repeat(10, '-');
```

it returns a string made up of `10` dashes, like this `"----------"`. It takes an integer count and a character, and it returns a string made up of that character repeated that many times.

Let us use this method in our `RectangleTest` class. In that class, we separated our test cases by outputting a sequence of dashes:

```
System.out.println("--------------------");
```

Replace this statement with the following statement:

```
System.out.println(Utility.repeat(21, '-'));
```

The IDE will flag the `Utility` class reference in red because it is not in the `RectangleTest` class and thus needs to be imported. If pressing `Alt-Enter` does not resolve this issue then you don't have the `i2c` library properly set up. Either it has not been downloaded properly or it has not been placed in `app/libs` or it has not been linked to this project as a dependency.

Re-run the test class and the output should be the same as before. It doesn't matter what character you use (dash, asterisk, equal sign, etc.) or what the repetition count is. All what we are after is a visual separation of the various test cases.

L0.2.d – Using APIs

In the previous section, we used a method provided by the `i2c` library, but in order to do so, we had to know:

- The name of the class in which the method resides
- The name of the method and the parameters it expects
- What the method does, not how it does it; i.e. what it returns

This information is given in the **A**pplication **P**rogramming **I**nterface (**API**) of the library. For `i2c`, you can find an overview of its API in Appendix-D and a full API in the website of this book. For the Java Standard Library, as well as the Android Platform Library, the API is built into our IDE and also available online—simply search for *Java API* or *Android API*.

You will become comfortable with APIs and learn how to use them as you progress through this book. In this section, we will take a look at a few examples, from both `i2c` and the Java libraries, focusing only at functionally-designed classes (aka *utility* classes) in which all methods are `static`.

The easiest way to explore a library is to write a test method that uses it. In the previous two sections, we learned how to write a Java class (such as `Rectangle`) and how to write a test class (such as `RectangleTest`). But in the case of libraries, the classes are already written, so we need only write a test class. Hence, create a test class (Section L0.2.b) named `LibTest` in the *zero* sub-package of the *Test* package.

The first method we would like to explore is `max`. It resides in the `Math` class of the Standard Java Library (built into the IDE). This method has the following API:

Methods (Math class)

```
public static int max(int a, int b)
```
Returns the greater of two given int values.

This tells us the `max` is `public` (and hence available for use from outside the library); that it is `static` (so we can invoke it using the `Math.max` syntax); and that it takes two integers and returns an integer. The return is the largest of the two parameters.

To explore this method and verify our understanding of its API, add a method to `LibTest`:

```
public class LibTest
{
    @Test
    public void maxTest()
    {
        int a, b;
        System.out.println("Testing Math.max:");
        a = 12; b = 3;
        System.out.println(a + "," + b);
        System.out.println(Math.max(a, b));
    }
}
```

Running this test leads to the following output:

```
Testing Math.max:
12,3
12
```

As a second example, let us explore the `gcd` method in the `Utility` class of `i2c`. Its API is shown below:

Methods (Utility class)

```
public static int gcd(int a, int b)
```
Return the GCD (Greatest Common Divisor) of the two passed integers.
Throws an exception if either integer is not positive.

We see that this is also a `public static` method so we can invoke it using `Utility.gcd`. The method takes two integers and returns an integer equal to their Greatest Common Divisor, GCD. The API also indicates that the two parameters are expected to be positive or else an exception will be thrown; i.e. the program will crash.

To explore this method, add a method to `LibTest`:

```
public class LibTest
{
    @Test
    public void maxTest()
    {
        ... // as before
    }

    @Test
    public void gcdTest()
    {
        int a, b;
        System.out.println("Testing Utility.gcd:");
        a = 18; b = 24;
        System.out.println(a + "," + b);
        System.out.println(Utility.gcd(a, b));
    }
}
```

Run this test (by right-clicking the whole class or just the newly added method) and examine the test output. Did you get `6`? Is this the correct result?

As a third API example, let us explore the `pow` method in the `Math` class of the Standard Java Library (built into the IDE). Its API is shown, in part, below:

Methods (Math class)

`public static double pow(double b, double e)`
Returns the value of the base `b` raised to the power of the exponent `e`.

The `pow` method computes powers. It takes two real numbers (real in Java is `double`), being the base and the exponent, and returns the power as a double. Hence, `Math.pow(2.0,3.0)` returns `8.0` because 2^3 = `2*2*2 = 8`. Similarly, `Math.pow(9.0,0.5)` returns `3.0`.

To explore this method, add a third method to `LibTest`:

```
public class LibTest
{
    @Test
    public void maxTest()
    {
        ... // as before
    }

    @Test
    public void gcdTest()
    {
        ... // as before
    }

    @Test
    public void powTest()
    {
        double a, b;
        System.out.println("Testing Math.pow:");

        a = 2.0; b = 3.0;
        System.out.println(a + "," + b);
        System.out.println(Math.pow(a, b));

        a = 9.0; b = 0.5;
        System.out.println(a + "," + b);
        System.out.println(Math.pow(a, b));

        a = 2.0; b = 0.5;
        System.out.println(a + "," + b);
        System.out.println(Math.pow(a, b));
    }
}
```

As before, we start with the declaration of the variables and then output a line that indicates what is being tested. For each test case, we output it along with the method's return. The test cases we chose are 2^3, $\sqrt{9}$, and $\sqrt{2}$.

If you run this test by right-clicking the newly added method and selecting *Run*, you should see the output of the three test cases as shown below:

```
Testing Math.pow:
2.0,3.0
8.0

9.0,0.5
3.0

2.0,0.5
1.4142135623730951
```

(We added an empty line between the test cases for clarity. You can optionally incorporate this separator into the method as we did in L0.2.b.)

Using APIs to solve problems (rather than developing everything from scratch) has become a hallmark of modern software construction. A typical software system today is nothing but an ensemble of a large number of components, mostly from libraries and few built from scratch specifically for the system at hand. For example, a typical, medium-size mobile app contains tens of thousands of methods but of these, only a hundred or so are written from scratch. This miniscule ratio (of custom-built to ready-made components) enables us to tackle problems that would be formidable were we to build everything from scratch.

The success of component-based software construction relies on the components being able to "talk to each other". Hence, the ability to read and understand APIs and the contracts of the methods (what each requires in order to work, and what it delivers post completion) has become as important a skill as developing code from scratch.

Note: Oracles and Assertions

We found that running `@Test` methods enables us to inspect the return of any method, and this works whether the method resides in a class that we wrote (such as `Rectangle`), or in a library we downloaded (such as i2c), or in a built in library (such as Java's). But how do we know that the return is correct? In the above `powTest` for example, we see the return of the first test case is 8, but is it correct? To answer this question, we need to find an **oracle** that can provide a trusted answer from a different source, e.g. a calculator. Given the oracle's answer, we simply **assert** that the return should be equal to it. If the assertion is met, the test passes else it fails. We will pursue this strategy in the coming chapters.

L0.3 – THE CS TRAIL

L0.3.a – About: Roadmap

This trail appears at the end of every learning chapter in the book. It takes you on a journey through some of the foundational concepts of Computer Science. What you will learn along the trail is Android independent; it is in fact agnostic to technology.

Here are some of the lookout points along the trail:

- **Computability**
  Decidability, Complexity, and P vs NP
- **Abstraction**
  Indirection, Delegation, and Recursion
- **I/O**
  Console, GUI, Parameter/Return, and Network

Note that the trail is cyclic: it revisits the same concept more than once, but at an increasing depth with every encounter.

L0.3.b – I/O: Console

We found in Section L0.1.a that algorithms transform information, and as such, an algorithm always starts with some information, known as its *input*, and ends with information known as its *output*.

The **I**nput / **O**utput (**I/O**) of an algorithm takes different forms depending on the source of the input and the destination of the output. We normally think of input as keystrokes on a keyboard and of output as text on the screen. This is indeed the traditional way for programs to communicate with the end user, and we will refer to it as **console** I/O. You use this mode whenever you work in the **CLI** (**C**ommand **L**ine **I**nterface) environment, such as the *Terminal* application in MacOS, Linux, and Unix; and the so-called *Command Prompt* program in Windows.

Programmatically, we can perform console output using a statement like this:

```
System.out.println( something );
```

We can also do console input using the `Scanner` class.

But console I/O is not the only possible mode: input can, for example, originate from the movement of a mouse in a GUI environment. It can also originate from a sensor on a device such as an accelerometer. Similarly, output need not be on the screen but rather on a file or an actuator such as a voice synthesizer. And more abstractly, input (to a method) can be implemented through received parameters and output through the return.

We will explore these modes, along with file and network I/O in the rest of this book.

L0.4 – EXERCISES

1. Computer Science is rooted in Math so it is not surprising that all its terms have precise meanings. The following two phrases, for example, are synonymous in everyday language but they have distinct meanings in CS:

   - *A list of integers.*
   - *A set of integers.*

   What is the difference between the two?

2. We use the word "bracket" often in everyday life but different people associate different meanings to it. The rigour in CS does not tolerate such ambiguity. Consider the three-column table shown below

bracket	{ }	less than / greater than
parenthesis	< >	array element
brace	[ ]	method signature
angle bracket	( )	block / scope

   The first column shows four terms, the second four pairs of symbols, and the third contains four Java element names. Associate each term with its corresponding symbol and corresponding Java element.

3. Provide an everyday life example in which abstraction layers (Section L0.1.b) are used to make a task easier.

4. Consider the task of following a recipe to make a cake. In one recipe, we use eggs, flour, sugar, baking powder, baking soda, and vanilla plus frosting. In a second recipe, we use a ready-made cake mix plus frosting.

   a) Which recipe is easier to follow?
   b) Is abstraction involved here?

5. Add the following method to the `Rectangle` class:

```
public static double getDiagonal(int width, int height)
```

It is supposed to return the length of the diagonal of the given rectangle.

Recall that by the Pythagorean theorem, the sought length is the square root of the sum of the squares of the rectangle's width and height. Use the `pow` method of the `Math` class to compute the square root.

6. Add the following method to the `RectangleTest` class:

```
@Test
public void getDiagonalTest()
```

Implement it so that it tests the `getDiagonal` method of `Rectangle` using at least two test cases. Use your calculator as oracle to determine if the method passes the test or not.

7. Re-implement the `getDiagonal` method of `Rectangle` but using the **`sqrt`** method of the `Math` class instead of its `pow` method. Its API is shown below:

Methods (Math class)
`public static double sqrt(double x)` Returns the positive square root of a given double value.

Re-run the test and ensure the answers are the same for all test cases.

8. Add the following method to the `LibTest` class:

```
@Test
public void factorialTest()
```

Implement it to test the following `i2c` method:

Methods (Utility class)
`public static double factorial(int n)` Determine the factorial `n!` of the passed integer. The factorial of a positive integer `n` is the product of all integers in `[1,n]`. Throws an exception if `n` is negative.

Include at least three test cases and use your calculator as oracle. Have you tried `n=0`? How about a negative `n`?

9. Add the following method to the `LibTest` class:

```
@Test
public void gfTest()
```

Implement it to test the following `i2c` method:

Methods (Utility class)
`public static int gf(int x)` Determine the Greatest Factor (GF) of the passed integer. A factor of a positive integer x is an integer in [1,x) that divides x evenly.

Include the following test cases in your implementation:

- `x = 12`
- `x = 47`
- `x = 102`
- `x = -10`

Run the test and inspect the output.
You should get the following results: `6`, `1`, `51`, exception.

10. Add the following method to the `LibTest` class:

```
@Test
public void m2FtInchTest()
```

Implement it so it tests the **`m2FtInch`** method of the `Utility` class of the `i2c` library. The API of that method is shown below:

Methods (Utility class)
`public static String m2FtInch(double h)` Convert a length measured in meters to feet and inches. The returned string is formatted as `F'I"` format, where `F` is the number of feet and `I` is the number of inches rounded to the nearest integer.

Do you understand what the method does? To verify, add a few test cases and view the output. Use a calculator, or an online converter, to verify. As an example, the return of `m2FtInch(1.78)` should be the string `5'10"`.