Week 2 - Unit Testing

The primary objective of software testing is to identify and eliminate errors in the code.

There are many software testing methodologies, techniques, and tools. CMSC 115 introduces two basic testing approaches:

- **Black-box Testing** assesses program functionality based on requirements of what the program should do, without looking into the details of how the program is implemented. To perform black-box testing in CMSC 115, we will develop test cases that map a program input to an expected output, or map a method call to an expected return value.
- **White Box Testing** assesses program functionality based on its implementation. To perform white-box testing in CMSC 115, we will develop test cases to cover various execution paths through a program.

The week#2 projects will be tested using black-box testing techniques. Each test case will specify:

- the user input
- the expected output based on the requirements
- the actual output based on the program execution
- the test result of pass/fail

Example #1 - Convert Inches to Feet

Consider the following program requirements:

Write a program that converts inches to feet. The program should read in an integer representing the number of inches, then calculate and display the equivalent number of feet. There are 12 inches in a foot.

The table below contains 5 test cases that will be used for testing. The first three columns can be filled out based on the specified program requirements. The last two columns are filled out by executing the program and observing the actual output.

Test	Input	Expected Output	Actual Output	Pass/Fail
1	3	3 inches = 0.25 feet		
2	12	12 inches = 1.0 feet		
3	18	18 inches = 1.5 feet		
4	24	24 inches = 2.0 feet		
5	33	30 inches = 2.75 feet		

The `InchesToFeet` class represents a possible solution:

```java
import java.util.Scanner;

/**
 * InchesToFeet reads the number of inches from user input
 * and prints the equivalent number of feet.
 */
public class InchesToFeet {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter inches: ");
        int inches = input.nextInt();
        double feet = inches / 12;
        System.out.println(inches + " inches = " + feet + " feet");
    }
}
```

`InchesToFeet` should be executed for each test case to obtain the actual output and determine the pass/fail result as shown below. While the tests for 12 and 24 inches pass, the other tests fail to produce the correct result.

Test	Input	Expected Output	Actual Output	Pass/Fail
1	3	3 inches = 0.25 feet	3 inches = 0.0 feet	Fail
2	12	12 inches = 1.0 feet	12 inches = 1.0 feet	Pass
3	18	18 inches = 1.5 feet	18 inches = 1.0 feet	Fail
4	24	24 inches = 2.0 feet	24 inches = 2.0 feet	Pass
5	33	33 inches = 2.75 feet	33 inches = 2.0 feet	Fail

The failed tests indicate an error exists in the `InchesToFeet` class. Recall how division works in Java. If both operands are integers, the result is an integer and any remainder is discarded. However, if either operand is a floating point number, the result is a floating point number.

Expression	Value
18 / 12	1
18 / 12.0	1.5

The code should be updated to use 12.0 as the divisor to avoid integer division:

```java
double feet = inches / 12.0;
```

After updating the code, the program is re-executed for each test case to confirm the actual output matches the expected output:

Test	Input	Expected Output	Actual Output	Pass/Fail
1	3	3 inches = 0.25 feet	3 inches = 0.25 feet	Pass
2	12	12 inches = 1.0 feet	12 inches = 1.0 feet	Pass
3	18	18 inches = 1.5 feet	18 inches = 1.5 feet	Pass
4	24	24 inches = 2.0 feet	24 inches = 2.0 feet	Pass
5	33	33 inches = 2.75 feet	33 inches = 2.75 feet	Pass

Example #2 - Converting years to minutes

Consider the following program requirements:

Write a program that converts years to minutes. The program should read in an integer representing the number of years, then calculate and display the equivalent number of minutes. Assume a year has 365 days, a day has 24 hours, and an hour has 60 minutes.

The requirements do not specify a particular range of values for years, other than stating the input is an integer. We'll use the following test cases:

Test	Input	Expected Output	Actual Output	Pass/Fail
1	1	525600 minutes		
2	2	1051200 minutes		
3	1000	525600000 minutes		
4	5000	2628000000 minutes		

The `YearsToMinutes` class represents a possible solution:

```java
import java.util.Scanner;

/**
 * YearsToMinutes reads the number of years from user input
 * and prints the equivalent number of minutes.
 */
public class YearsToMinutes {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the number of years: ");
        int years = input.nextInt();
        int minutes = years * 365 * 24 * 60;
        System.out.println(minutes + " minutes");
    }
}
```

We'll run `YearsToMinutes` for each test case. Notice the last test fails to produce the expected result.

Test	Input	Expected Output	Actual Output	Pass/Fail
1	1	525600 minutes	525600 minutes	Pass
2	2	1051200 minutes	1051200 minutes	Pass
3	1000	525600000 minutes	525600000 minutes	Pass
4	5000	2628000000 minutes	-1666967296 minutes	Fail

An **overflow** occurs when a calculation produces a result that is too large for the declared data type of a variable.

- An `int` is a 32-bit signed integer data type that can store whole numbers ranging from: -2,147,483,648 to 2,147,483,647. Java provides constants `Integer.MIN_VALUE` and `Integer.MAX_VALUE` for these values.
- The last test case results in an overflow error because `minutes` is declared as an `int` and can't store a value as large as 2,626,000,000.

A `long` can store a value as large as 9,223,372,036,854,775,807. Assume the code is updated to declare the variable as `long`:

```
long minutes = years * 365 * 24 * 60;
```

After updating the code, the program is re-executed for each test case. Unfortunately, the last test case still fails as shown below:

Test	Input	Expected Output	Actual Output	Pass/Fail
1	1	525600 minutes	525600 minutes	Pass
2	2	1051200 minutes	1051200 minutes	Pass
3	1000	525600000 minutes	525600000 minutes	Pass
4	5000	2628000000 minutes	-1666967296 minutes	Fail

Although the variable `minutes` on the left-hand side of the assignment is declared as a long, the expression on the right-hand side produces an int:

```
long minutes = years * 365 * 24 * 60;
```

Why does this happen? Since the variable `years` is declared as an `int`, the expression `years * 365 * 24 * 60` produces an `int` and results in an overflow error. The solution is to either cast `years` as a long, or to specify one of the numeric literal values as a long by append `L`.

Expression	Type
years * 365	int
(long) years * 365	long
years * 365L	long

We'll update the code to cast the `years` variable as a `long`. Note this does not modify the actual type of the variable `years`, rather it creates a temporary copy of the value stored in memory as a long.

```
long minutes = (long) years * 365 * 24 * 60;
```

Executing the updated program for each test case results in success:

Test	Input	Expected Output	Actual Output	Pass/Fail
1	1	525600 minutes	525600 minutes	Pass
2	2	1051200 minutes	1051200 minutes	Pass
3	1000	525600000 minutes	525600000 minutes	Pass
4	5000	2628000000 minutes	2628000000 minutes	Pass