NYP LIT

C FAMILY WORKSHOP

Programming with C#

14 June 2023

Agenda

1) Introduction to the C# Family

2) Introduction to .NET

3) Getting Started with C#

4) Hands On Time

Introduction to the C family

Consists mainly of C, C++, C#

Back to Agenda Page

C

What is C?

- General-purpose language that provides low-level access to system memory
- Mainly used for OSes, databases, language compilers

Pros

- Structured-programming approach
- Dynamic memory allocation

Cons

- No exception handling
- Not compatible for OOP

C++

What is C++?

- Cross-platform improvement of C
- Mainly used for OOP, supports procedural and functional programming

Pros

- Notable portability
- Despite being a high-level language, it can do low-level manipulation

Cons

- No garbage collection
- Hard to use pointers

C#

What is C#?

- General purpose programming language
- Used on the Windows .NET framework
- Versatile and new language used for OOP

Pros

- Statically typed and easy to read
- Scalable and easy to maintain

Cons

- Needs to be compiled every time a change is made
- .NET applications can only be run on Windows

Introduction to .NET

Back to Agenda Page

.NET Framework and Class Library

- A computing platform developed by Microsoft that simplifies application development in a highly distributed environment
- Provides commonly used and reusable classes
- Ensures that all programmers have access to common classes

.NET Framework and Class Library

- A virtual execution system called the common language runtime (CLR)
- Implementation by Microsoft of the common language infrastructure (CLI)
- CLI is the basis for creating execution and development environments

Getting started with C#

Brief history, features, data types and variable

Back to Agenda Page

A brief history of C#

- Developed in the year 2000 by Microsoft's Anders Hejlsberg, a Danish software engineer
- Originally titled COOL, a clever acronym that stood for "C-like Object Oriented Language"
- Originally designed to rival java, it gained popularity quickly among new and seasoned developers

Prominent features of C#

Garbage Collection

Reclaims memory occupied by unreachable and unused objects

Nullable Types

Guard against variables that don’t refer to allocated objects

Lambda Expressions

Support functional programming techniques

Prominent features of C#

Language Integrated Query

Syntax that creates common pattern

Language Support (Asynchronous)

Syntax for building distributed system

Primitive Types

1) `Int`

Any whole number between approximately -2,000,000,000 to 2,000,000,000

2) `Bool`

Either True or False

3) `Double`

Any number with decimal point

Primitive Types

4) Char

Any single character, digit, and/or punctuation mark.

Enclosed in single-quotation (‘) marks.

5) String

Zero or more characters, digits, and/or punctuation marks.

Enclosed in quotation (”) marks.

Data types and classes in C#

- C# makes use of a unified type system
- All C# types such as `int` and `double`, are inherited from a single root `object` type
- Provides iterators, which enable implementers of collection classes to define custom behaviors for client code

Types and Variables Structures

- A type defines the structure and behavior of any data in C#
- The declaration of a type may include:
  - its members
  - base type
  - interfaces it implements
  - operations permitted for that type

E.g. A variable is a label that refers to an instance of a specific type.

C# Versioning

- Ensure programs and libraries evolve over time in a compatible manner
- Aspects of C#'s design that were directly influenced by versioning considerations include:
  - separate `virtual` and `override` modifiers
  - the rules for method overload resolution
  - support for explicit interface member declarations

Elements of C# code

C# Hands On

Back to Agenda Page

Expressions in C#

- A combination of operands (variables, literals, method calls) and operators that can be evaluated to a single value
- An expression must have at least one operand but may not have any operator.

For example:

```
double temperature;
temperature = 42.05;
```

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

Statements in C#

- A basic unit of execution of a program
- A program consists of multiple statements

For example:

```
int age = 21;
```

```
int marks = 90;
```

Declaration Statements

- Used to declare and initialize variables

  For example:

```
char ch;
```

```
int maxValue = 55;
```

Expression Statements

- An expression followed by a semicolon is called an expression statement

  For example:

```
/* Assignment */
area = 3.14 * radius * radius;
/* Method call is an expression*/
System.Console.WriteLine("Hello");
```

Expression Statements

Selection Statements

if...else, switch

Iteration Statements

do, while, for, foreach

Expression Statements

Jump Statements

break, continue, goto, return, yield

Exception Handling Statements

throw, try-catch, try-finally,
try-catch-finally

Blocks in C#

- A combination of zero or more statements that is enclosed inside curly brackets { }

```
double temperature = 42;
if (temperature > 32)
{   //Start of Block
    Console.WriteLine("Curent temperature = {0}", temperature);
    Console.WriteLine("it's hot");
}   //End of Block
```

Variables in C#

`int`

Stores integers (whole numbers), without decimals.

`double`

Stores floating point numbers, with decimals.

Variables in C#

`char`

Stores single characters. Surrounded by single quotes.

`string`

Stores text. Surround by double quotes.

`bool`

Stores values. True or False.

Declaring Variables in C#

- To create a variable, you must specify the type and assign it a value;
  - `type variableName = value;`

- Where `type` is a C# type (such as `int` or `string`), and `variableName` is the name of the variable. The equal sign is used to assign values to the variable.

Displaying Variables in C#

- The `WriteLine()` method is often used to display variable values to the console window. To combine both text and a variable, use the + character.

```
string name = "Shermaine";
Console.WriteLine("Hello " + name);
```

Declaring Constants in C#

- If you don't want to overwrite existing values, add the **`const`** keyword in front of the variable type. This will declare the variable as "**constant**", which means unchangeable and read-only

```
const int myNum = 15;
myNum = 20; // Error:
```

Getting User Input in C#

- Just as `Console.WriteLine()` is used to output (print) values, Now we will use `Console.ReadLine()` can be used to get user input.

Getting User Input in C#

- The `Console.ReadLine()` method returns a string. Therefore, you cannot get information from another data type, such as `int`. The following program will cause an error:

```
Console.WriteLine("Enter your age:");

int age = Console.ReadLine();

Console.WriteLine("Your age is: " + age);
```

BREAK

Back to Agenda Page

Typecasting in C#

Type casting is when you assign a value of one data type to another type. In C#, there are two types of casting:

- Implicit Casting (automatically) - converting a smaller type to a larger type size
  - `char -> int -> long -> float -> double`
- Explicit Casting (manually) - converting a larger type to a smaller size type
  - `double -> float -> long -> int -> char`

Implicit Casting

- Implicit casting is done automatically when passing a smaller size type to a larger size type:

```
int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

Console.WriteLine(myInt); // Outputs 9

Console.WriteLine(myDouble);  // Outputs 9
```

Explicit Casting

- Explicit casting must be done manually by placing the type in parentheses in front of the value:

```
double myDouble = 9.78;

int myInt = (int) myDouble;       // casting double -> int

Console.WriteLine(myDouble);      // Outputs 9.78

Console.WriteLine(myInt);         // Outputs 9
```

Arithmetic Operators in C#

Operators	Name	Description	Example
+	Addition	Adds together two values	1 + 1 == 2
-	Subtraction	Subtracts one value from another	2 - 1 == 1
*	Multiplication	Multiplies two values	1 * 2
/	Division	Divides one value by another	4 / 2 == 2
%	Modulus	Returns the division remainder	3 % 2 == 1
++	Increment	Increases the value of a variable by 1	x = 0 x ++ //x == 1
--	Decrement	Decreases the value of a variable by 1	x – //x == 0

Comparison Operators in C#

Operators	Name	Example
==	Equal to	x == y
!=	Not Equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or Equal to	x >= y
<=	Less than or Equal to	x <= y

Logical Operators in C#

Operators	Name	Description	Example
&&	Logical and	Returns True if both statements are true	x < 5 && x < 10
||	Logical or	Returns True if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns False if the result is true	!(x < 5 && x < 10)

`Strings` in C#

- Strings are used for storing text. A string is a collection of characters surrounded by double quotes:

```
string greeting = "Hello";
```

- In C#, strings are object, which have properties and methods. For example, the length of a string can be found with the `Length` property:

```
string txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

Console.WriteLine("The length of the txt string is: " + txt.Length);
```

- string methods include `ToUpper()` and `ToLower()`, which returns a copy of the string converted to uppercase or lowercase:

```
string txt = "Hello World";

Console.WriteLine(txt.ToUpper());   // Outputs "HELLO WORLD"

Console.WriteLine(txt.ToLower());   // Outputs "hello world"
```

`String` Concatenation in C#

- The `+` operator can be used between `strings` to combine them. This is called concatenation:

```
string firstName = "John ";
string lastName = "Lim";
string name = firstName + lastName;
Console.WriteLine(name);
```

- You can also use the `string.Concat()` method to concatenate two strings:

```
string firstName = "John ";
string lastName = "Lim";
string name = string.Concat(firstName, lastName);
Console.WriteLine(name);
```

Booleans in C#

- Very often, in programming, you will need a data type that can only have one of two values, like:
  - YES / NO
  - ON / OFF
  - TRUE / FALSE
  - For this, C# has a `bool` data type, which can take the values `true` or `false`.
- Example:

```
bool isCSharpFun = true;
bool isFishTasty = false;
Console.WriteLine(isCSharpFun);   // Outputs true
Console.WriteLine(isFishTasty);   // Outputs false
```

Conditionals in C#

- C# has the following conditional statements:
  - Use `if` to specify a block of code to be executed, if a specified condition is true
  - Use `else` to specify a block of code to be executed, if the same condition is false
  - Use `else if` to specify a new condition to test, if the first condition is false
  - Use `switch` to specify many alternative blocks of code to be executed

`else` statements in C#

- Use the `else` statement to specify a block of code to be executed if the condition is False.

```
int time = 20;
if (time < 18) {
  Console.WriteLine("Good day.");
}
else {
  Console.WriteLine("Good evening.");
}  // Outputs "Good evening."
```

`else if` statements in C#

- Use the `else if` statement to specify a new condition if the first condition is False.

```
int time = 22;
if (time < 10) {
  Console.WriteLine("Good morning.");
}
else if (time < 20) {
  Console.WriteLine("Good day.");
}
else {
  Console.WriteLine("Good evening.");
}  // Outputs "Good evening."
```

`Switch` statements in C#

- Use the `switch` statement to select one of many code blocks to be executed.
  - The `switch` expression is evaluated once
  - The value of the expression is compared with the values of each case
  - If there is a match, the associated block of code is executed
  - The `break` and `default` keywords will be described later in this chapter

```
int day = 4;
switch (day) {
  case 1:
    Console.WriteLine("Monday");
    break;
  case 2:
    Console.WriteLine("Tuesday");
    break;
  case 3:
    Console.WriteLine("Wednesday");
    break;
  case 4:
    Console.WriteLine("Thursday");
    break;
  case 5:
    Console.WriteLine("Friday");
    break;
  case 6:
    Console.WriteLine("Saturday");
    break;
  case 7:
    Console.WriteLine("Sunday");
    break;
}     // Outputs "Thursday" (day 4)
```

`While` statements in C#

- The `while` loop loops through a block of code as long as a specified condition is True:
- Syntax:

```
while (condition)
{
  // code block to be executed
}
```

- In the example below, the code in the loop will run, over and over again, as long as a variable (`i`) is less than `5`:

```
int i = 0;

while (i < 5)

{

 Console.WriteLine(i);

 i++;

}
```

Variant of `while` statements in C#

- The `do/while` loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Syntax:

```
do
{
  // code block to be executed
}
while (condition);
```

`For` loops in C#

- When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop:
- Syntax:

```
for (statement 1; statement 2; statement 3)
{
  // code block to be executed
}
```

- `Statement 1` is executed (one time) before the execution of the code block. `Statement 2` defines the condition for executing the code block. `Statement 3` is executed (every time) after the code block has been executed. The example below will print the numbers 0 to 4:

```
for (int i = 0; i < 5; i++)
{
  Console.WriteLine(i);
}
```

`Foreach` loops in C#

- There is also a `foreach` loop, which is used exclusively to loop through elements in an array:
- Syntax:

```
foreach (type variableName in arrayName)
{
  // code block to be executed
}
```

Break in C#

- The `break` statement can be used to jump out of a loop. This example jumps out of the loop when i is equal to 4:
- Example:

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    break;
  }
  Console.WriteLine(i);
}
```

Continue in C#

- The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop. This example skips the value of 4:
- Example:

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  Console.WriteLine(i);
}
```

Arrays in C#

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value. To declare an array, define the variable type with square brackets:
  - `string[] cars;`
- We have now declared a variable that holds an array of strings. To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces:
  - `string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};`
- To create an array of integers, you could write:
  - `int[] myNum = {10, 20, 30, 40};`

Accessing elements of an array in C#

- You access an array element by referring to the index number. This statement accesses the value of the first element in `cars`:

```
string[] cars = {"Volvo", "BMW", "Ford",
"Mazda"};

Console.WriteLine(cars[0]);

// Outputs Volvo
```

Changing an array element in C#

- To change the value of a specific element, refer to the index number:

```
string[] cars = {"Volvo", "BMW", "Ford",
"Mazda"};

cars[0] = "Opel";

Console.WriteLine(cars[0]);

// Now outputs Opel instead of Volvo
```

Array length in C#

- To find out how many elements an array has, use the `Length` property:

```
string[] cars = {"Volvo", "BMW", "Ford",
"Mazda"};

Console.WriteLine(cars.Length);

// Outputs 4
```

Lists in C#

- `List<T>` is a class that contains multiple objects of the same data type that can be accessed using an index. For example:

```
// list containing integer values

List<int> number = new List<int>() { 1, 2, 3 };
```

- Here, `number` is a `List` containing integer values (1, 2 and 3).

- To create `List<T>` in C#, we need to use the `System.Collections.Generic` namespace. Here is how we can create `List<T>`. For example:

```
using System.Collections.Generic;
class Program {
    public static void Main() {
        // create a list named subjects
that contain 2 elements
        List<string> subjects = new
List<string>() { "English", "Math" };
    }
}
```

QUIZ TIME! <3

Quiz link:
https://www.gimkit.com/

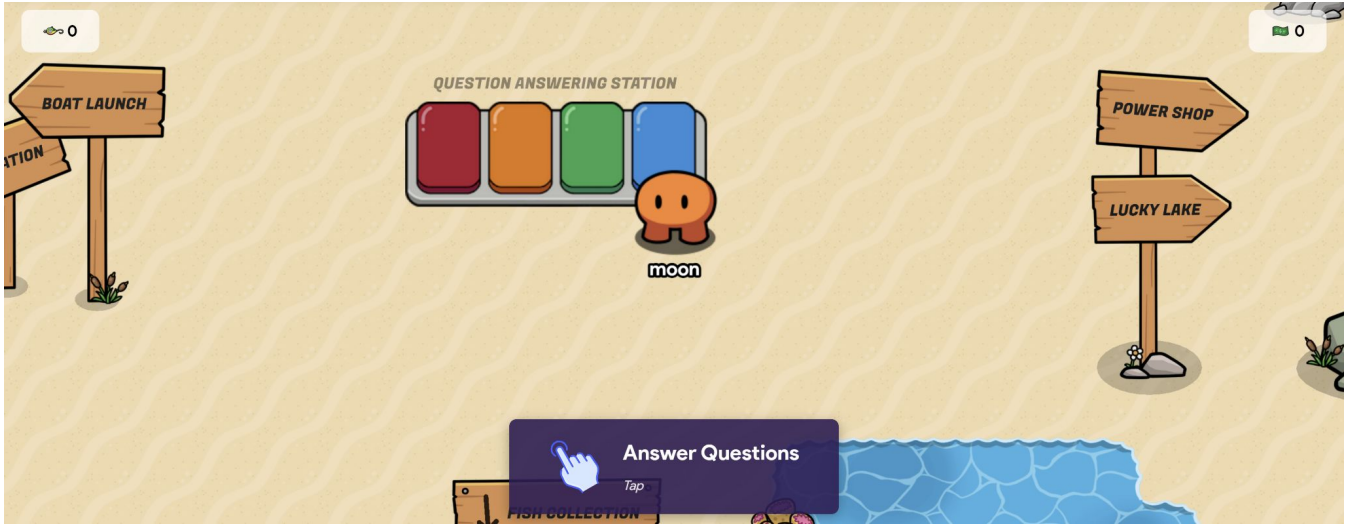
Back to Agenda Page

A guide on how to use the GimKit quiz interface:

- Once the quiz starts, you might find yourself on a page looking similar to this:

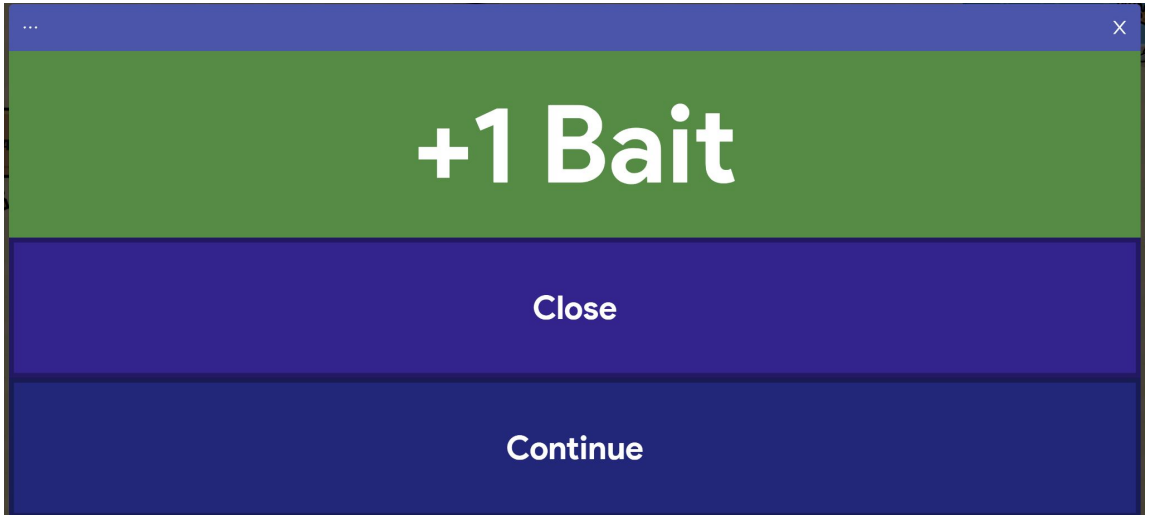
A guide on how to use the GimKit quiz interface:

- Navigate your way through to this position to start answering questions:



A guide on how to use the GimKit quiz interface:

- As you answer questions, the number of baits that you own will increase:



A guide on how to use the GimKit quiz interface:

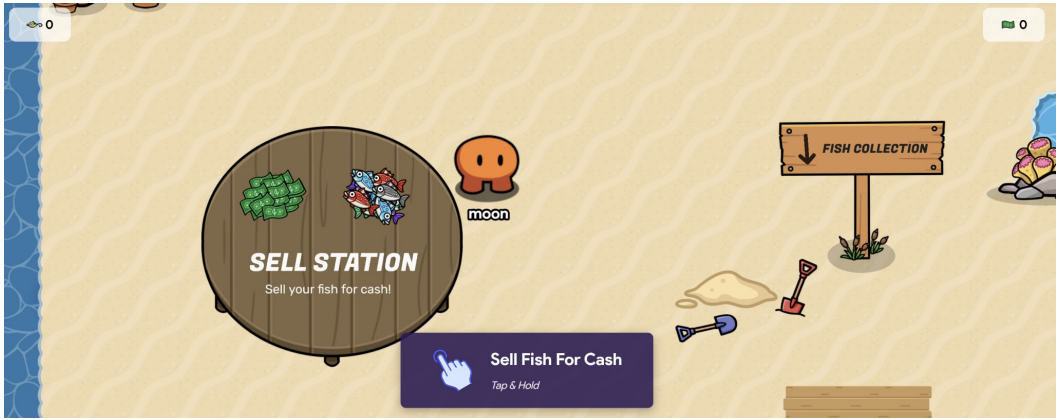
- After answering correct questions, you can close the question-answer pop-up, and navigate to this position to cast a fishing rod in order to catch fish:

A guide on how to use the GimKit quiz interface:

- Now, you have caught yourself a fish. Close this pop-up after you have spent all of your baits:

A guide on how to use the GimKit quiz interface:

- Navigate to this position to sell your fish and earn cash. Each fish can earn you different amounts of cash. The winners will be decided based on the amount of cash you have earned in 10 minutes.



QR code for quiz

Dictionaries in C#

- A `Dictionary<TKey, TValue>` is a generic collection that consists of elements as key/value pairs that are not sorted in an order. For example:

```
Dictionary<int, string> country = new
Dictionary<int, string>();
```

- Here, `country` is a dictionary that contains `int` type keys and `string` type values.

- To create a dictionary in C#, we need to use the `System.Collections.Generic` namespace. Here is how we can create a dictionary in C#.

```
dictionaryName = new Dictionary<dataType1, dataType2>();
```

- Here:
- `dictionaryName` - name of the dictionary
- `dataType1` - datatype of keys
- `dataType2` - datatype of values

Example:

```
using System.Collections;
class Program {
    public static void Main() {
        // create a dictionary
        Dictionary<int, string> country = new Dictionary<int,
string>();

        // add items to dictionary
        country.Add(5, "Brazil");
        country.Add(3, "China");
        country.Add(4, "Usa");

        // print value having key is 3
        Console.WriteLine("Value having key 3: " +
country[3]);
    }
}
```

Ternary operators in C#

- There is also a short-hand if else, which is known as the ternary operator because it consists of three operands. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements:

```
variable = (condition) ? expressionTrue :
expressionFalse;
```

Instead of writing:

```
int time = 20;

if (time < 18) {

    Console.WriteLine(
"Good day.");

} else {

    Console.WriteLine(
    "Good evening.");

}
```

We can simplify it to:

```
int time = 20;

int time = 20;

string result = (time < 18)
? "Good day."

: "Good evening.";

Console.WriteLine(result);
```

Methods in C#

- A `method` is a block of code which only runs when it is called. You can pass data, known as parameters, into a method. Methods are used to perform certain actions, and they are also known as functions. Why use methods? To reuse code: define the code once, and use it many times.

- A method is defined with the name of the method, followed by parentheses `()`. C# provides some pre-defined methods, such as `Main()`, but you can also create your own methods to perform certain actions:

```
class Program{
  static void Main(string[] args) {
    // code to be executed
  }
}
```

- `Main()` is the name of the method
- `static` means that the method belongs to the Program class and not an object of the Program class.
- `void` means that this method does not have a return value.

- To call (execute) a method, write the method's name followed by two parentheses () and a semicolon; In the following example, `MyMethod()` is used to print a text (the action), when it is called:

```
static void MyMethod() {
  Console.WriteLine("I just got
executed!");
}
static void Main(string[] args){
  MyMethod();
}   // Outputs "I just got executed!"
```

Hands On Time!

Back to Agenda Page

Exercise

- Create a Program that checks if the number that the user inputs is a prime number
- It should continue asking the user for numbers to check until the user enters an invalid input
- Use `int.TryParse(input, out int num)` to convert `string` to `int`
  - `input` is the `string` representing the `int`
  - `int.TryParse` will `return true` if it successfully converts `input` to `int`, `false` if otherwise
  - The parsed `int` value will be assigned to `num`
- Bonus: Do it recursively

```
class Program {
   static void Main(string[] args) {
       int num;
       while (true) {
           Console.WriteLine("Enter a number (1 or less to exit): ");
           if (int.TryParse(Console.ReadLine(), out num)) {
               if (num <= 1) { break; }
               else { Console.WriteLine((isPrime(num))? "Prime" : "Not Prime"); }
           } else {
               Console.WriteLine("Invalid input");
           }
       }
   }
   static bool isPrime(int num, int divisor = 2) {
       if (num == divisor) {
           //Prime numbers are only divisible by 1 and themselves;
           return true;
       } else if (num % divisor == 0) {
           //If no remainder(divisible), is not prime
           return false;
       } else {
           //If not divisible, try the next one
           return isPrime(num, divisor + 1);
       }
   }
}
```

Thank You

Please help us feel in this feedback form here: