Understanding Data Types

Test your Knowledge

1. **What type would you choose for the following “numbers”?**

Value	Recommended Data Type
Person’s telephone number	`string`
Person’s height	`double`
Person’s age	`int`
Person’s gender	`string`
Person’s salary	`decimal`
Book’s ISBN	`string`
Book’s price	`decimal`
Book’s shipping weight	`double`
Country’s population	`long`
Number of stars in the universe	`BigInteger`
Employees in a business	`int`

2. **What are the difference between value type and reference type variables? What is boxing and unboxing?**

Difference Between Value Type and Reference Type in C#

In C#, data types are categorized into Value Types and Reference Types, depending on how they store and manage data in memory.

1) Value Type

- Stored in stack memory, which provides faster access.
- Holds the actual data directly rather than a reference.
- Created using structures (struct), enums, and primitive types (int, float, bool, etc.).
- Cannot accept null unless declared as a nullable type (int?).
- Automatically removed when they go out of scope.

2) Reference Type

- Stored in heap memory, which is managed by the Garbage Collector.
- Stores a reference (memory address) to the actual data, rather than the data itself.
- Created using classes (class), interfaces, delegates, and arrays.
- Can accept null values.
- Manually removed by the Garbage Collector when no longer referenced.

What is Boxing?

Definition: Boxing is the process of converting a value type into a reference type by wrapping it inside an object or an interface.

Characteristics:

Happens implicitly.
Moves data from stack memory to heap memory.
Used when a value type needs to be stored as an object.

Example of Boxing:

```
int num = 10;  // Value type (stored in stack)
object obj = num;  // Boxing: num is now stored in heap as an object
Console.WriteLine(obj); // Output: 10
```

What is Unboxing?

Definition: Unboxing is the process of converting a reference type (object) back to a value type.

Characteristics:

Happens explicitly (requires casting).
Moves data from heap memory to stack memory.
Used when retrieving a value type stored as an object.
Example of Unboxing:

```
object obj = 10;  // Boxing (stored in heap)
int num = (int)obj;  // Unboxing (explicit conversion back to int)
Console.WriteLine(num); // Output: 10
```

3. **What is meant by the terms managed resource and unmanaged resource in .NET**

Managed Resource and Unmanaged Resource in .NET

1) Managed Resource:

- A resource that is handled automatically by the .NET runtime (CLR) and Garbage Collector (GC).
- Includes memory allocated for .NET objects such as string, List<T>, class instances.
- No manual cleanup is required.

2) Unmanaged Resource:

- A resource not managed by the .NET runtime, requiring explicit cleanup.
- Includes file handles, database connections, network sockets, and COM objects.
- Needs manual release using IDisposable.Dispose() or finalizers (~Destructor) to prevent memory leaks.

4. **Whats the purpose of Garbage Collector in .NET**

Purpose of Garbage Collector (GC) in .NET

The Garbage Collector (GC) in .NET is responsible for automatically managing memory by reclaiming unused objects and preventing memory leaks.

Key Purposes of Garbage Collector:

1. Automatic Memory Management → Frees memory occupied by objects that are no longer in use.

2. Prevents Memory Leaks → Ensures unused objects don’t consume system resources.
3. Optimizes Performance → Reduces manual memory management overhead, allowing efficient execution.
4. Manages Object Lifetime → Determines when objects are no longer accessible and removes them.
5. Handles Heap Allocation → Allocates and deallocates memory dynamically for managed objects.

How it Works:

- Objects are allocated in the Managed Heap.
- GC periodically identifies and removes unreachable objects.
- Memory is compacted to optimize storage and avoid fragmentation.

Practice number sizes and ranges

1. Create a console application project named /02UnderstandingTypes/ that outputs the number of bytes in memory that each of the following number types uses, and the minimum and maximum values they can have: sbyte, byte, short, ushort, int, uint, long, ulong, float, double, and decimal.

```
internal class PrintSize

{

    public static void Main(string[] args)

    {

        Console.WriteLine("Data Type Information in C#:");

        Console.WriteLine("----------------------------------------");


        PrintTypeInfo("sbyte", sizeof(sbyte), sbyte.MinValue, sbyte.MaxValue);

        PrintTypeInfo("byte", sizeof(byte), byte.MinValue, byte.MaxValue);

        PrintTypeInfo("short", sizeof(short), short.MinValue, short.MaxValue);

        PrintTypeInfo("ushort", sizeof(ushort), ushort.MinValue, ushort.MaxValue);

        PrintTypeInfo("int", sizeof(int), int.MinValue, int.MaxValue);

        PrintTypeInfo("uint", sizeof(uint), uint.MinValue, uint.MaxValue);

        PrintTypeInfo("long", sizeof(long), long.MinValue, long.MaxValue);

        PrintTypeInfo("ulong", sizeof(ulong), ulong.MinValue, ulong.MaxValue);

        PrintTypeInfo("float", sizeof(float), float.MinValue, float.MaxValue);

        PrintTypeInfo("double", sizeof(double), double.MinValue, double.MaxValue);

        PrintTypeInfo("decimal", sizeof(decimal), decimal.MinValue, decimal.MaxValue);


        Console.WriteLine("\nPress any key to exit...");
```

```
        Console.ReadKey();
    }

    static void PrintTypeInfo(string typeName, int size, object min, object max)
    {
        Console.WriteLine(typeName.PadRight(10) + " | Size: " + size + " bytes | Min: " +
min + " | Max: " + max);
    }
}
```

2. Write program to enter an integer number of centuries and convert it to years, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds. Use an appropriate data type for every data conversion. Beware of overflows!

Input: 1

Output: 1 centuries = 100 years = 36524 days = 876576 hours = 52594560 minutes

= 3155673600 seconds = 3155673600000 milliseconds = 3155673600000000

microseconds = 3155673600000000000 nanoseconds

Input: 5

Output: 5 centuries = 500 years = 182621 days = 4382904 hours = 262974240

minutes = 15778454400 seconds = 15778454400000 milliseconds = 15778454400000000

microseconds = 15778454400000000000 nanoseconds

```
internal class TimeConvert
{
    public static void Main(string[] args)
    {
        Console.Write("Enter number of centuries: ");
        int centuries = int.Parse(Console.ReadLine());


        int years = centuries * 100;
        int days = years * 36524 / 100;  // 1 century = 36524 days (accounting for leap years)
        long hours = days * 24;
        long minutes = hours * 60;
        long seconds = minutes * 60;
```

```
        long milliseconds = seconds * 1000;

        long microseconds = milliseconds * 1000;

        ulong nanoseconds = (ulong)microseconds * 1000; // Use ulong to prevent
overflow



        Console.WriteLine($"{centuries} centuries = {years} years = {days} days = {hours}
hours = {minutes} minutes = {seconds} seconds = {milliseconds} milliseconds =
{microseconds} microseconds = {nanoseconds} nanoseconds");



    }

}
```

Controlling Flow and Converting Types

Test your Knowledge

1. **What happens when you divide an int variable by 0?**

   Dividing an int by 0 in C# throws a DivideByZeroException at runtime, causing the program to crash unless handled with a try-catch block.

   Example:

   ```
   int a = 10, b = 0;
   int result = a / b; // Throws DivideByZeroException
   ```

2. **What happens when you divide a double variable by 0?**

   It does not throw an exception. Instead:
   double.PositiveInfinity if positive (10.0 / 0).
   double.NegativeInfinity if negative (-10.0 / 0).
   double.NaN (Not-a-Number) if 0.0 / 0.0.

3. **What happens when you overflow an int variable, that is, set it to a value beyond its range?**

   When you overflow an int variable (exceeding int.MaxValue or going below int.MinValue), the behavior depends on whether overflow checking is enabled:

   Unchecked Context (Default Behavior)

   The value wraps around (circular behavior).

   Example:

   ```
   int num = int.MaxValue;
   num += 1;
   Console.WriteLine(num); // Output: -2147483648 (wraps around)
   ```

Checked Context (OverflowException)

If checked is used, an OverflowException is thrown.

Example:

```
checked
{
   int num = int.MaxValue;
   num += 1; // Throws OverflowException
}
```

4. **What is the difference between x = y++; and x = ++y;?**

   x = y++; (Post-increment) → Assigns y to x, then increments y.
   x = ++y; (Pre-increment) → Increments y, then assigns it to x.

   Example:

```
int y = 5;
int x = y++; // x = 5, y = 6
int z = ++y; // y = 7, z = 7
```

5. **What is the difference between break, continue, and return when used inside a loop statement?**

   break → Exits the loop completely.
   continue → Skips the current iteration and moves to the next.
   return → Exits the method entirely, stopping the loop and function.

   Example:

```
for (int i = 1; i <= 5; i++)
{
   if (i == 3) continue; // Skips when i == 3
   if (i == 4) break;    // Stops loop when i == 4
   Console.WriteLine(i);
}
```

Output:

Copy code
1
2

6. **What are the three parts of a for statement and which of them are required?**

Initialization (optional) → int i = 0;
Condition (required) → i < 10;
Iteration (optional) → i++
Example of a valid for loop without initialization or iteration:

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

7. **What is the difference between the = and == operators?**

= → Assignment operator (assigns value).
== → Equality operator (checks if values are equal).
Example:

```
int x = 5;   // Assignment
bool check = (x == 5); // Equality check (true)
```

8. **Does the following statement compile? for (; true;) ;**

Yes, it compiles and creates an infinite loop that does nothing.

9. **What does the underscore _ represent in a switch expression?**

_ acts as a default case (matches anything not handled explicitly).

Example:

```
int number = 5;
string result = number switch
{
    1 => "One",
    2 => "Two",
    _ => "Other" // Default case
};
Console.WriteLine(result); // Output: Other
```

10. **What interface must an object implement to be enumerated over by using the foreach statement?**

The object must implement IEnumerable or IEnumerable<T>.
Example:

```
class MyCollection : IEnumerable<int>
{
    private List<int> numbers = new() { 1, 2, 3 };
    public IEnumerator<int> GetEnumerator() => numbers.GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
}
```

Practice loops and operators

1. FizzBuzzis a group word game for children to teach them about division. Players take turnsto count incrementally, replacing any number divisible by three with the word /fizz/, any number divisible by five with the word /buzz/, and any number divisible by both with / fizzbuzz/.
   Create a console application in Chapter03 named Exercise03 that outputs a simulated FizzBuzz game counting up to 100. The output should look something like the following
   Screenshot:

```
internal class Exercise03
{
    public static void Main(string[] args)
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i % 3 == 0 && i % 5 == 0)
                Console.WriteLine("FizzBuzz");
            else if (i % 3 == 0)
                Console.WriteLine("Fizz");
            else if (i % 5 == 0)
                Console.WriteLine("Buzz");
            else
                Console.WriteLine(i);
        }
    }
}
```

What will happen if this code executes?

```
Int max = 500 ;

for(bytei=0;i<max;i++)
{
WriteLine(i);
}
```

Create a console application and enter the preceding code. Run the console application and view the output. What happens?

Issue:

byte can only hold values from 0 to 255.
When i reaches 255, it overflows to 0 instead of reaching 500.
The loop becomes infinite because i < max is always true.

What code could you add (don’t change any of the preceding code) to warn us about the problem?

Add a checked block to detect overflow:

```
checked
{
   for (byte i = 0; i < max; i++)
   {
      Console.WriteLine(i);
   }
}
```

This will throw an OverflowException instead of looping indefinitely.

2. Print-a-Pyramid.Like the star pattern examples that we saw earlier, create a program that will print the following pattern: If you find yourself getting stuck, try recreating the two examples that we just talked about in this chapter first. They're simpler, and you can compare your results with the code included above. This can actually be a pretty challenging problem, so here is a hint to get you going. I used three total loops. One big one contains two smaller loops. The bigger loop goes from line to line. The first of the two inner loops prints the correct number of spaces, while the second inner loop prints out the correct number of stars.

```
    *
   ***
  *****
 *******
*********
```

```
internal class Print_a_Pyramid
{
    static void Main()
    {
        int rows = 5; // Change this to adjust pyramid height

        for (int i = 1; i <= rows; i++)
        {
            // Print spaces
            for (int j = 1; j <= rows - i; j++)
                Console.Write(" ");

            // Print stars
            for (int k = 1; k <= (2 * i - 1); k++)
                Console.Write("*");

            Console.WriteLine(); // Move to the next line
        }
    }
}
```

3. Write a program that generates a random number between 1 and 3 and asks the user to guess what the number is. Tell the user if they guess low, high, or get the correct answer. Also, tell the user if their answer is outside of the range of numbers that are valid guesses (less than 1 or more than 3). You can convert the user's typed answer from a string to an int using this code:

   int guessedNumber = int.Parse(Console.ReadLine());

   Note that the above code will crash the program if the user doesn't type an integer value.
   For this exercise, assume the user will only enter valid guesses.

```
internal class NumberGuessing
{
   static void Main()
   {
      Random random = new Random();
      int correctNumber = random.Next(1, 4); // Generates 1, 2, or 3

      Console.Write("Guess a number (1-3): ");
      int guessedNumber = int.Parse(Console.ReadLine());

      if (guessedNumber < 1 || guessedNumber > 3)
         Console.WriteLine("Invalid guess! Please enter a number between
1 and 3.");
      else if (guessedNumber < correctNumber)
         Console.WriteLine("Too low!");
      else if (guessedNumber > correctNumber)
         Console.WriteLine("Too high!");
      else
         Console.WriteLine("Correct! You guessed it!");
   }
}
```

4. Write a simple program that defines a variable representing a birth date and calculates how many days old the person with that birth date is currently. For extra credit, output the date of their next 10,000 day (about 27 years) anniversary.
   Note: once you figure out their age in days, you can calculate the days until the next anniversary using int daysToNextAnniversary = 10000 - (days % 10000);
   .

```
internal class BirthDayCalculator
{
    static void Main()
    {
        // Define birthdate (Change this to your birthdate)
        DateTime birthDate = new DateTime(1995, 1, 1); // Example: Jan 1, 1995

        // Get today's date
        DateTime today = DateTime.Today;

        // Calculate age in days
        int daysOld = (today - birthDate).Days;

        // Calculate next 10,000-day anniversary
        int daysToNextAnniversary = 10000 - (daysOld % 10000);
        DateTime nextAnniversary = today.AddDays(daysToNextAnniversary);

        // Display results
        Console.WriteLine($"You are {daysOld} days old.");
        Console.WriteLine($"Your next 10,000-day anniversary is on {nextAnniversary:MMMM dd, yyyy}");
    }
}
```

5. Write a program that greets the user using the appropriate greeting for the time of day. Use only if , not else or switch , statements to do so. Be sure to include the following
greetings:
"Good Morning"
"Good Afternoon"
"Good Evening"
"Good Night"
It's up to you which times should serve as the starting and ending ranges for each of the greetings. If you need a refresher on how to get the current time, see DateTime Formatting. When testing your program, you'll probably want to use a DateTime variable you define, rather than the current time. Once you're confident the program works correctly, you can substitute DateTime.Now for your variable (or keep your variable and just assign DateTime.Now as its value, which is often a better approach)

```
internal class Greeting
{
    static void Main()
    {
        // Use DateTime.Now for real-time greeting
        DateTime currentTime = DateTime.Now;
        int hour = currentTime.Hour; // Get the current hour

        // Determine the appropriate greeting using only if statements
        if (hour >= 5 && hour < 12)
            Console.WriteLine("Good Morning");
        if (hour >= 12 && hour < 17)
            Console.WriteLine("Good Afternoon");
        if (hour >= 17 && hour < 21)
            Console.WriteLine("Good Evening");
        if ((hour >= 21 && hour <= 23) || (hour >= 0 && hour < 5))
            Console.WriteLine("Good Night");

        // Display current time (optional)
        Console.WriteLine($"Current Time: {currentTime:hh:mm tt}");
    }
}
```

6. Write a program that prints the result of counting up to 24 using four different increments.
   First, count by 1s, then by 2s, by 3s, and finally by 4s.
   Use nested for loops with your outer loop counting from 1 to 4. You inner loop should count from 0 to 24, but increase the value of its /loop control variable/ by the value of the / loop control variable/ from the outer loop. This means the incrementing in the / afterthought/ expression will be based on a variable.

   Your output should look something like this:

   0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24
   0,2,4,6,8,10,12,14,16,18,20,22,24
   0,3,6,9,12,15,18,21,24
   0,4,8,12,16,20,24

```
internal class IncrementalCounter
{
    static void Main()
    {
        for (int i = 1; i <= 4; i++) // Outer loop (increment values: 1, 2, 3, 4)
        {
            for (int j = 0; j <= 24; j += i) // Inner loop (counts with step i)
            {
                Console.Write(j);
                if (j + i <= 24) Console.Write(","); // Add comma except for last
number
            }
            Console.WriteLine(); // New line after each sequence
        }
    }
}
```