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

* A person’s telephone number: String
* A person’s height: float or double
* A person’s age: int
* A person’s gender (Male, Female, Prefer Not To Answer): enum
* A person’s salary: decimal
* A book’s ISBN: String
* A book’s price: decimal
* A book’s shipping weight: float or double
* A country’s population: long
* The number of stars in the universe: bigInteger
* The number of employees in each of the small or medium businesses in the United Kingdom (up to about 50,000 employees per business): int

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

**Value Types:**

Storage: Value types are stored in the stack.

Contains Data: Value type variables directly contain their data.

Memory Allocation: Memory is allocated at compile time.

Performance: Access to value types is generally faster since they are stored in the stack.

Examples: int, float, bool, char, struct, enum.

**Reference Types:**

Storage: Reference types are stored in the heap, but the reference (or pointer) to the data is stored in the stack.

Contains Reference: Reference type variables contain a reference (or pointer) to the actual data.

Memory Allocation: Memory is allocated at runtime.

Performance: Access to reference types is generally slower due to additional dereferencing.

Examples: class, string, array, delegate, interface.

**Boxing**: Implicit conversion from a value type to an object. It involves allocating memory on the heap and copying the value from the stack to the heap.

**Unboxing**: Explicit conversion from an object to a value type. It involves checking the object type and copying the value from the heap back to the stack.

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

**Managed Resources**: Handled by the .NET garbage collector, no need for explicit cleanup.

**Unmanaged Resources**: Not handled by the garbage collector, require explicit cleanup via Dispose and finalizers to avoid resource leaks.

4. What’s the purpose of Garbage Collector in .NET?

**Automatic memory management**: Reduces the need for manual memory management by developers.

**Memory reclamation**: Frees up memory occupied by unreferenced objects.

**Prevention of memory leaks**: Ensures efficient use of memory resources.

**Memory optimization**: Compacts the heap to reduce fragmentation and improve allocation performance.

**Generational collection**: Optimizes collection performance based on object lifespan.

**Finalization**: Handles resource cleanup through finalizers.

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.

using System;

namespace UnderstandingTypes

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Type: sbyte");

Console.WriteLine($"Size: {sizeof(sbyte)} bytes");

Console.WriteLine($"Min: {sbyte.MinValue}");

Console.WriteLine($"Max: {sbyte.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: byte");

Console.WriteLine($"Size: {sizeof(byte)} bytes");

Console.WriteLine($"Min: {byte.MinValue}");

Console.WriteLine($"Max: {byte.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: short");

Console.WriteLine($"Size: {sizeof(short)} bytes");

Console.WriteLine($"Min: {short.MinValue}");

Console.WriteLine($"Max: {short.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: ushort");

Console.WriteLine($"Size: {sizeof(ushort)} bytes");

Console.WriteLine($"Min: {ushort.MinValue}");

Console.WriteLine($"Max: {ushort.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: int");

Console.WriteLine($"Size: {sizeof(int)} bytes");

Console.WriteLine($"Min: {int.MinValue}");

Console.WriteLine($"Max: {int.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: uint");

Console.WriteLine($"Size: {sizeof(uint)} bytes");

Console.WriteLine($"Min: {uint.MinValue}");

Console.WriteLine($"Max: {uint.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: long");

Console.WriteLine($"Size: {sizeof(long)} bytes");

Console.WriteLine($"Min: {long.MinValue}");

Console.WriteLine($"Max: {long.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: ulong");

Console.WriteLine($"Size: {sizeof(ulong)} bytes");

Console.WriteLine($"Min: {ulong.MinValue}");

Console.WriteLine($"Max: {ulong.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: float");

Console.WriteLine($"Size: {sizeof(float)} bytes");

Console.WriteLine($"Min: {float.MinValue}");

Console.WriteLine($"Max: {float.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: double");

Console.WriteLine($"Size: {sizeof(double)} bytes");

Console.WriteLine($"Min: {double.MinValue}");

Console.WriteLine($"Max: {double.MaxValue}");

Console.WriteLine();

Console.WriteLine("Type: decimal");

Console.WriteLine($"Size: {sizeof(decimal)} bytes");

// Decimal.MinValue and Decimal.MaxValue are constants in System.Decimal

Console.WriteLine($"Min: {decimal.MinValue}");

Console.WriteLine($"Max: {decimal.MaxValue}");

Console.WriteLine();

}

}

}

//output:

Type: sbyte

Size: 1 bytes

Min: -128

Max: 127

Type: byte

Size: 1 bytes

Min: 0

Max: 255

Type: short

Size: 2 bytes

Min: -32768

Max: 32767

Type: ushort

Size: 2 bytes

Min: 0

Max: 65535

Type: int

Size: 4 bytes

Min: -2147483648

Max: 2147483647

Type: uint

Size: 4 bytes

Min: 0

Max: 4294967295

Type: long

Size: 8 bytes

Min: -9223372036854775808

Max: 9223372036854775807

Type: ulong

Size: 8 bytes

Min: 0

Max: 18446744073709551615

Type: float

Size: 4 bytes

Min: -3.4028235E+38

Max: 3.4028235E+38

Type: double

Size: 8 bytes

Min: -1.7976931348623157E+308

Max: 1.7976931348623157E+308

Type: decimal

Size: 16 bytes

Min: -79228162514264337593543950335

Max: 79228162514264337593543950335

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

using System;

namespace CenturiesConverter

{

class Program

{

static void Main(string[] args)

{

// Prompt the user to enter the number of centuries

Console.Write("Enter the number of centuries: ");

if (!int.TryParse(Console.ReadLine(), out int centuries) || centuries < 0)

{

Console.WriteLine("Please enter a valid non-negative integer.");

return;

}

// Conversion constants

const int YearsPerCentury = 100;

const int DaysPerYear = 365; // Using 365.24 would be more accurate but we'll stick with integer values

const int DaysPerLeapYear = 366;

const double AverageDaysPerYear = 365.24; // More accurate average

// Calculations

int years = centuries \* YearsPerCentury;

int days = (int)(years \* AverageDaysPerYear);

long hours = days \* 24L;

long minutes = hours \* 60L;

long seconds = minutes \* 60L;

long milliseconds = seconds \* 1000L;

long microseconds = milliseconds \* 1000L;

long nanoseconds = microseconds \* 1000L;

// Output results

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

}

}

}

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

When you divide an **int** variable by 0, a **DivideByZeroException** is thrown at runtime. Division by zero is undefined for integers, and attempting to do so results in this exception.

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

When you divide a **double** variable by 0, it does not throw an exception. Instead, it results in either positive infinity (**Infinity**), negative infinity (**-Infinity**), or **NaN** (Not a Number) depending on the dividend:

* If the dividend is a positive number, the result is positive infinity.
* If the dividend is a negative number, the result is negative infinity.
* If the dividend is zero, the result is **NaN**.

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

When an **int** variable overflows (exceeds its maximum or minimum value), it wraps around to the other end of its range. For example, if an **int** exceeds **int.MaxValue**, it wraps around to **int.MinValue**, and vice versa. This behavior can be checked and controlled using the **checked** and **unchecked** keywords:

* In a **checked** context, an **OverflowException** is thrown.
* In an **unchecked** context, the value wraps around without any exception.

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

* **x = y++;** (Post-increment): The value of **y** is assigned to **x**, and then **y** is incremented by 1.
* **x = ++y;** (Pre-increment): **y** is incremented by 1 first, and then the new value of **y** is assigned to **x**.

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

* **break**: Exits the nearest enclosing loop or switch statement immediately.
* **continue**: Skips the remaining statements in the current iteration of the nearest enclosing loop and proceeds to the next iteration.
* **return**: Exits the method in which it appears, and returns control to the calling method. If **return** is used inside a loop, it terminates the loop and the entire method execution.

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

1. **Initialization**: This part is executed once at the beginning and is used to initialize the loop control variable(s).
2. **Condition**: This part is evaluated before each iteration. If the condition is **true**, the loop body executes; if **false**, the loop terminates.
3. **Iteration**: This part is executed after each iteration of the loop body and is used to update the loop control variable(s).

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

* **= (Assignment operator)**: Assigns the value on the right to the variable on the left.
* **== (Equality operator)**: Compares the values on both sides for equality and returns **true** if they are equal, **false** otherwise.

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

Yes, the statement compiles. It represents an infinite loop because the condition **true** is always true. The loop body is empty due to the semicolon (**;**) immediately following the loop declaration.

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

In a switch expression, the underscore **\_** acts as a discard pattern and represents the default case, matching any value not matched by previous cases. It is similar to a **default** case in a traditional **switch** statement.

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

An object must implement the **IEnumerable** interface (or **IEnumerable<T>** for generic collections) to be enumerated over by using the **foreach** statement. This interface provides the **GetEnumerator** method, which returns an enumerator that iterates through the collection.

FizzBuzz Code:

using System;

namespace Exercise03

{

class Program

{

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);

}

}

}

}

}

int max = 500;

for (byte i = 0; i < max; i++)

{

Console.WriteLine(i);

}

This code will result in an infinite loop and an eventual overflow of the **byte** variable **i**.

1. **byte Data Type**: The **byte** data type in C# is an unsigned 8-bit integer with a range of 0 to 255.
2. **Loop Condition**: The loop's condition checks if **i** is less than **max** (which is 500).
3. **Overflow**: Since **byte** can only hold values from 0 to 255, once **i** exceeds 255, it wraps around back to 0 (byte overflow). Therefore, **i** will never reach 500, causing an infinite loop.

**Corrected Code**

If you want to print numbers from 0 to 499, you should use an **int** or **short** type instead of **byte** for the loop counter:

int max = 500;

for (int i = 0; i < max; i++)

{

Console.WriteLine(i);

}

Guess user input between 1-3:

using System;

class Program

{

static void Main()

{

// Generate a random number between 1 and 3

int correctNumber = new Random().Next(3) + 1;

Console.WriteLine("Guess the number (between 1 and 3):");

// Read user input and convert it to an integer

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

// Check if the guessed number is within the valid range (1 to 3)

if (guessedNumber >= 1 && guessedNumber <= 3)

{

// Provide feedback based on the user's guess

if (guessedNumber < correctNumber)

{

Console.WriteLine("You guessed too low.");

}

else if (guessedNumber > correctNumber)

{

Console.WriteLine("You guessed too high.");

}

else

{

Console.WriteLine("Congratulations! You guessed the correct number.");

}

}

else

{

Console.WriteLine("Your guess is outside of the valid range (1 to 3).");

}

}

}

Print pyramids:  
using System;

class Program

{

static void Main()

{

int rows = 5; // Number of rows for the pyramid

// Outer loop for each row

for (int i = 1; i <= rows; i++)

{

// Inner loop for spaces before stars

for (int j = 1; j <= rows - i; j++)

{

Console.Write(" ");

}

// Inner loop for printing stars

for (int k = 1; k <= 2 \* i - 1; k++)

{

Console.Write("\*");

}

// Move to the next line after each row

Console.WriteLine();

}

}

}

Birthday:

using System;

class Program

{

static void Main()

{

// Define the birth date (replace with actual birth date)

DateTime birthDate = new DateTime(1995, 1, 15);

// Calculate the age in days

TimeSpan age = DateTime.Today - birthDate;

int daysOld = (int)age.TotalDays;

// Output the current age in days

Console.WriteLine($"You are {daysOld} days old.");

// Calculate days until the next 10,000 day anniversary

int daysToNextAnniversary = 10000 - (daysOld % 10000);

// Calculate the date of the next 10,000 day anniversary

DateTime nextAnniversary = DateTime.Today.AddDays(daysToNextAnniversary);

// Output the date of the next 10,000 day anniversary

Console.WriteLine($"Your next 10,000 day anniversary will be on: {nextAnniversary:d}");

}

}

Customer greetings:

using System;

class Program

{

static void Main()

{

// Define a DateTime variable for testing purposes

DateTime currentTime = new DateTime(2024, 5, 28, 22, 21, 25);

// Determine the appropriate greeting based on the time of day

int currentHour = currentTime.Hour;

string greeting;

if (currentHour >= 0 && currentHour < 6)

{

greeting = "Good Night";

}

if (currentHour >= 6 && currentHour < 12)

{

greeting = "Good Morning";

}

if (currentHour >= 12 && currentHour < 18)

{

greeting = "Good Afternoon";

}

if (currentHour >= 18 && currentHour < 24)

{

greeting = "Good Evening";

}

// Output the appropriate greeting

Console.WriteLine($"{greeting}, user!");

}

}

Count from 1-24:

using System;

class Program

{

static void Main()

{

// Outer loop controls the increment value

for (int increment = 1; increment <= 4; increment++)

{

// Inner loop counts from 0 to 24 with the specified increment

for (int i = 0; i <= 24; i += increment)

{

Console.Write(i);

// Add comma and space except after the last number in the line

if (i != 24)

{

Console.Write(", ");

}

}

Console.WriteLine(); // Move to the next line after each set of numbers

}

}

}