Data Types and Operations

C# Programming

In this chapter we will learn ...

- Understand main C# data types
- Data Type Operations
- Type Conversions

Name some C# Data Types?

Question

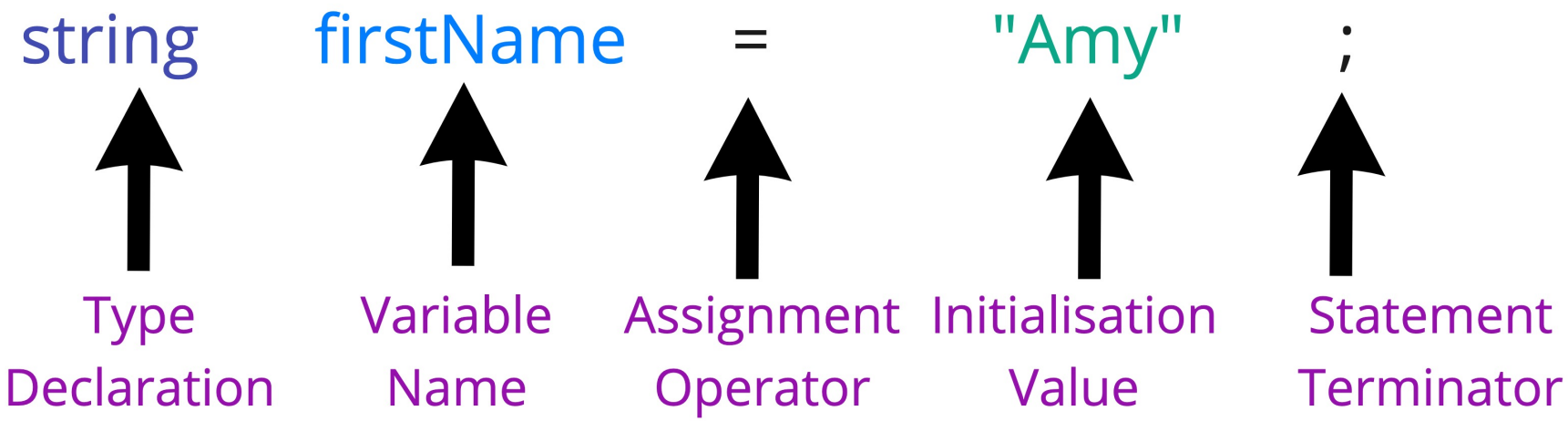
C# Language Data Types

C# specifies a number of builtin types

- int
- string, char
- float, double
- decimal
- bool

Act as aliases for the actual framework data types

Declaring Variables



Framework Data Types

C# Keyword	.NET Type	Size	Min	Max
sbyte	System.SByte	Signed 8 Bit Integer	-128 == 0-(Max + 1)	127
byte	System.Byte	Un-Signed 8 Bit Integer	0	255
short	System.Int16	Signed 16 Bit Integer	0 - (Max + 1)	32767
ushort	System.UInt16	Un-Signed 16 Bit Integer	0	65535
uint	System.UInt32	Un-Signed 32 Bit Integer	0	4294967295
int	System.Int32	Signed 32 Bit Integer	0 - (Max + 1)	2147483647
ulong	System.UInt64	Un-Signed 64 Bit Integer	0	18446744073709551615
long	System.Int64	Signed 64 Bit Integer	0 - (Max + 1)	9223372036854775807

C# or Framework Data Types

Most developers use C# language types

- int == System.Int32
- string == System.String

Code is more flexible

- Data Types can be redefined as framework changes

Integers

Whole Numbers

```
// 8 Bit Integers
System.SByte signed_8Bit_Int = 127; // Max == 127 Min = -128
System.Byte  unSigned_8Bit_Int = 255; // Max = 255 Min = 0
Console.WriteLine(signed_8Bit_Int);
Console.WriteLine(unSigned_8Bit_Int);
```

```
using System;
// Can declare Signed/UnSigned 16, 32, 64 bit integers
Int16  s16i = Int16.MaxValue;
UInt16 us16i = UInt16.MaxValue;
Console.WriteLine(s16i);  // 32767
Console.WriteLine(us16i); // 65535
```

Floating Point

Binary floating point types

- float - single precision - 32 bit
- double - double precision - 64 bit

Can suffer rounding/formatting issues

```
> 0.1 + 0.2 == 0.3
false
```

Decimal data type works better for curreny representation

Boolean

Simple Data Type for storing true or false

```
bool employed = true;
bool married = false;
if (employed == true && married)
{
    // Do Something
}
```

Character

Character Type defines a single character

```
char initial = 'C';
```

- Note the use of single quotes!

String

Standard C# strings are immutable (can't be changed in memory)

- Allocated onto the heap
- Accessed via a smart pointer called a reference

Assigning a new value to a string ...

- creates a new object on the heap
- re-adjusts the reference to point to the new object

Declaring a string

Use either declaration:

- both are equivalent although the first declaration is preferred

```
string s1;  // or
System.String s2;
```

String variables can be initialized to null at declaration

- No string object is created at this point

```
string s1 = null;
```

Creating Strings

Code below creates a blank string on the heap and 's' references it

```
string s = "";
```

Create a new string containing "Hello World"

- Blank string is left behind on the heap to be Garbage Collected

```
string s = "Hello World";
```

- Better would be because it doesn't leave a blank string to be GC'ed

```
string s = String.Empty;
```

Formatting Strings

String can be created from other data variables

```
string name = "Fred";
int age = 21;
string message = "Hello " + name + " you are " + age.ToString();
```

Or from interpolation using $ syntax

- Note the $ at the start of the string
- Variables are embedded inside the string within {}

```
string message = $"Hello {name} you are {age}";
```

Escaping Strings

Some string contain escape characters eg `\n`

```
string message = "Hello \n World";
Console.WriteLine(message);
```

Hello

World

Using @ in front of string switches of escape character interpretation

```
string message = @"Hello \n World";
Console.WriteLine(message);
```

Hello \n World

Raw string literals

Multiline strings can now be declared (C# 11)

- Place text between triple-double quotes """

```
string html= """
<html>
    <head></head>
    <body></body>
</html>

""";
```

Implicit Typing

C# can infer from a variable initialization what type a variable should be

- Use the **var** declaration to allow implicit typing
- Variable is still strongly typed once initialized (use dynamic to keep flexible)

```
int age = 21; // Explicityly typed as int

var name = "Beth"; // Implicitly typed as string
```

Type behaviors

.NET Data Types behavior depend on their classification

.NET Data Types can be

- Value Types
  - int, float, double, bool etc
  - structs, enums, tuples
- Reference Types
  - string, list etc
  - Anything defined as a class or record
- Pointer Types

Value Types - Quick Facts

- Allocated on the stack when declared with a function
- Defined by structs or enums
- Assigning one value type variable to another always creates a copy
- Do not need Garbage collection
    - (Can be attributes of an object which do)
- Don't support inheritance

Reference Types - Quick Facts

- Defined by two things
  - A reference variable that points to
  - On Object that is allocated on the heap
- Defined by classes
- Assigning one reference type variable to another only copies the reference NOT the object
- Need Garbage collection
- Support inheritance

Arrays - Collecting Data Together

Storing data in separate variable is not always practical

- We might want to process data together rather than separately
- Organizing data logically together can be useful

Arrays allow data of the same type to be arranged together

Arrays

Arrays are type-safe containers

- Inherit from System.Array
- Allocated onto the Heap
  - Accessed via references
- Indexed
- Usually 0 based and fixed size
- Elements initialized to type default
  - 0 for numbers, false for boolean
  - null for reference types

Types of Array

There are three types of array

- Single Dimension
- Multi Dimension
- Jagged

Single Dimension Array

Example: Array of five integers

```
int[] numbers = new int[5];
```

Could also be defined as

```
int[] numbers; // Define int Array reference variable
numbers = new int[5]; // Create array object and assign to reference variable
```

or ... define an array with predefined content (five zeros)

```
int[] numbers = {0, 0, 0, 0, 0};
```

Working with arrays

Members of an array can be accessed using indexes

- First element is at position 0
    - Last element is at n-1

```
> int[] numbers = {2, 4, 6, 8, 10};
> numbers
int[5] { 2, 4, 6, 8, 10 }
> numbers[1]
4
```

What would numbers[3] return?

Updating Arrays

Values can be assigned to individual array elements

- Specify the index for the element to update

Here we update the element as position 1

```
> int[] numbers = {2, 4, 6, 8, 10};
> numbers[1] = 99;
> numbers
int[5] { 2, 99, 6, 8, 10 }
```

- Because arrays are 0 based the second element in the array is updated

Types of operations

- Assignment operators
- Math operators
- Logical operators
- Bitwise operators
- Compound Operators

Assignment Operator

Most common operator to create data values

- Single **=** acts as assignment operator

```
string name = "Fred";
byte age = 21;
age = 33; // Assignment after declaration
```

Basic Operators

Operator	Description
+	Add
*	Subtract
/	Divide
%	Modulo Divide (remainder division)

```
> 2 + 3
5
> 5 - 3
2
> 2 * 4
8
```

What will be the result

```
> 10 / 3
```

- A. 3.3333333
- B. 3.0
- C. 3

Data Operation Rules

- Operations on like data types results in the same data type
  - operation on two integers will produce an integer result
- Operations on different data types will result in larger of two data types
  - byte and long -> long
  - int32 and decimal -> decimal

```
> 10 / 3
3
> 10 / 3.0
3.3333333333333335
```

Modulo Operator

% operator returns the remainder from integer division

```
> 10 / 3
3
> 10 % 3 // 10 == 3 * 3 == 9 remainder 1
1
```

Compound Operators

Basic Operators can be used in this way

```
> int count = 0;
> count = count + 1;
> count
1
```

+ operator can be used in a shortened/compound form

```
> count += 1; // Both syntax forms produce equivalent result
> count
2
```

Compound Operators

Here are the other compound operators

Operator	Name	Description
+=	Increment	Increase by x
-=	Decrement	Decrease by x
*=	Multiply	Multiply by x
/=	Divide	Divide by x
%=	Modulo	Modulo by x
<<= >>= &= ^= |=	Bitwise	Bitwise apply x (see later)

Unary Operators

Operator	Name	Description
++	Increment	Increase by 1
--	Decrement	Decrease by 1
-	Negate	Flip sign

```
> int count = 0;
> count++;
> count
1
```

Comparison Operators

```
x = 2;
```

Operator	Name	Examples
<	Less than	x<5 (returns true)
>	Greater than	x>5 (returns false)
<=	Less than equal to	x<=2 (returns true)
>=	Greater than equal to	x>=2 (returns true)
==	Equal equal to	x==2 (returns true)
!=	Not equal to	x!=2 (returns false)

Logical Operators

- Allow boolean results to be combined
  - Produce a boolean result
- Used as clauses in if, while, for, ternary statements

Operator	Name	Examples
&&	And	true && true -> true true && false -> false
| |	Or	true | | true -> true true | | false -> true
^	Xor	true ^ true -> false true ^ false -> true

Logical Operators - Example

- if statement with a logical clause

```
int age = 21;
string name = "Fred";

if (age < 18 && name.Length > 0) {
    Console.WriteLine("Valid Child's Name");
}
```

Question

- Does the C# runtime actually check the name length?

Short Circuit Operators

Logic Operators short circuit when combined

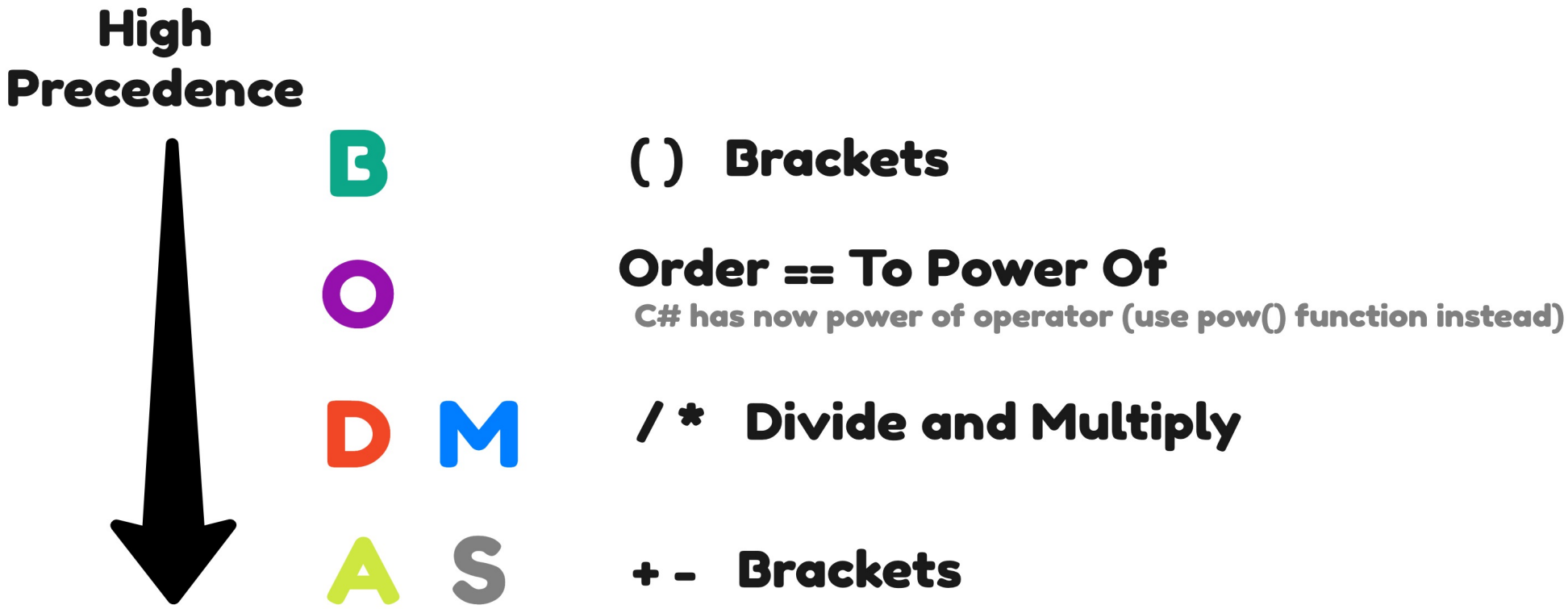
- && returns false if left clause evaluates false - does not test right clause!
- || returns true if left clause evaluates true - does not test right clause!

So given:

```
if (age < 18 && name.Length > 0) {
    Console.WriteLine("Valid Child's Name");
}
```

- age < 18 would short circuit
- name.Length > 0 would not be evaluated

Simple Precedence - BODMAS



BODMAS Example

- Precedence

```
> 10 + 5 * 3 // * has higher precedence than + and applies first
25
> (10 + 5) * 3 // () have higher precedence than * and apply first
45
```

- Associativity

```
> int b = 5, c = 1;
> int a = b = c; // = has right to left associativity
> a // op order is c = 1->b = c->a = b
1
```

Precedence - Decreasing Order

- Operators at the top of the list have highest order of precedence

Symbol	Type of operation	Associativity
[ ] ( ) . -> ++ -- (postfix)	Expression	Left to right
sizeof & * + - ~ ! ++ -- (prefix)	Unary	Right to left
typecasts	Unary	Right to left
* / %	Multiplicative	Left to right
+ -	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right

Precedence - Decreasing Order (continued)

Symbol	Type of operation	Associativity
== !=	Equality	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical-AND	Left to right
| |	Logical-OR	Left to right
? :	Conditional-expression	Right to left
= *= /= %= += -= <<= >>= &= ^= |=	Simple and compound assignment	Right to left

Type Conversion

Converting data types is a common need in programming

- How do we convert a string into an int
- How does a decimal get converted into a decimal?

We will look at

- Implicit Conversions
- Explicit Conversions
- Helper function Conversions

Implicit Conversions

Implicit conversions occur without any extra intervention and require

- Compatible Data Types
- Source Data Type must be <= memory size of destination type

```
UInt16 x = 10; // UInt16 is an unsigned 16 bit integer (Min value is 0)
UInt32 z = x; // Implicit conversion from UInt16 to UIn32
```

Big type, little type

What happens if you pour a bucket of water into a cup?

- It overflows!

Assigning large type variable to a small type variable is the same problem

If a 16 bit int assigned to an 8 bit int

- 8 bit variable is to small to receive all the bits
- Data bits will be lost!

Required Explicit Conversions

Assigning a smaller type to a larger (compatible) one requires "casting"

- To assign a bigger type to a smaller prefix variable with smaller type name inside brackets eg. `UInt16 smallNumber = (UInt16)bigNumber`

```
UInt32 z = UIInt16.MaxValue; // Assign the max value of a UInt16 to a UInt32
UInt16 x = (UInt16)z; // Notice the cast using the type we are assigning to in brackets
```

But ... We get an overflow error if the value in the larger type is too big for the small type to hold (Pouring a full bucket into a cup)

```
UInt32 z = UIInt32.MaxValue; // Assign the max value of a UInt16 to a UInt32
UInt16 x = (UInt16)z; // OVERFLOW!!!
```

Converting to Strings

Use ToString() on any variable to convert to a string

- Event custom data types can support ToString()

```
int age = 21;
string sAge = age.ToString();
```

Using the Convert class

You can use the Convert class and its static methods to

- Convert any data type to any other
- There are many To..() functions that facilitate conversions ToByte, ToChar, ToDecimal etc

```
> Convert.ToBoolean('H');
true
```

In this chapter we learned ...

- Understand main C# data types
- Data Type Operations
- Type Conversions