GRAND CIRCUS

C# BOOTCAMP

INTRO TO C#

ABOUT C#

C# .NET

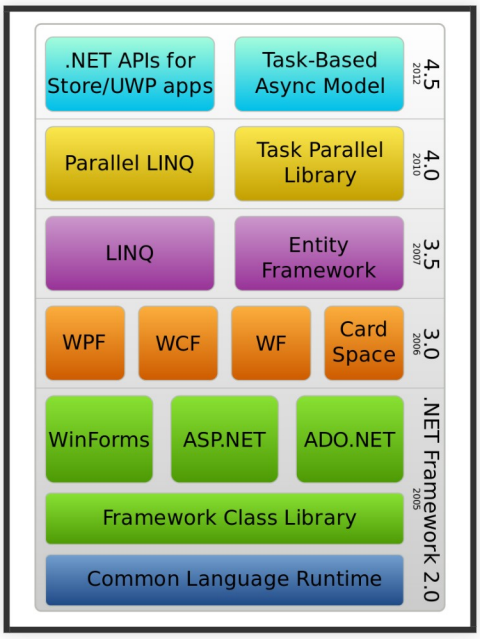
- Introduced by Microsoft back in 2000
- Object-Oriented
- Derives much of its syntax from C and C++, but has fewer low-level facilities than either of them
- Most recent version is 6.0

.NET FRAMEWORK

.NET framework is a software framework that is developed by Microsoft.

.NET is used to develop many applications including Web, Desktop, and Mobile Applications.

.NET FRAMEWORK



C# COMPARED TO JAVA AND C++

- C# is very similar to C++ and JAVA in syntax.
- C# and JAVA have a similar compilation architecture.

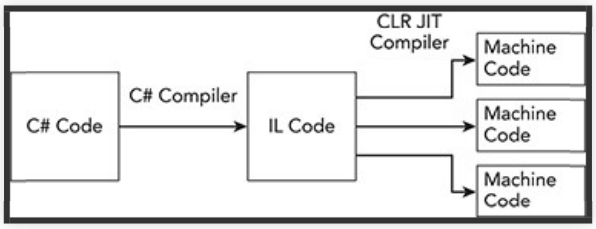
C# COMPARED TO JAVA AND C++

- C++ and C# run faster than Java, but Java is getting faster with each new version.
- Both Java and C# handle most memory operations automatically, while C++ programmers must write code that manages memory.

OPERATING SYSTEMS SUPPORTED BY C# .NET

- Windows
- Linux and OS X with the .NET MONO

HOW C# COMPILES AND INTERPRETS CODE



INTEGRATED DEVELOPMENT ENVIRONMENTS

Visual Studio is the Most Commonly used IDE for developing C#

Other IDEs include: SharpDevelop and MONODevelop

INTRO TO C#

DECLARING THE MAIN METHOD

```
public static void main(String[] args) {
//Statements
}
```

PRINTING DATA TO THE CONSOLE

We can use the Console class to do output:

Example:

```
ısole.WriteLine("Hello");
```

GETTING INPUT FROM THE CONSOLE

Getting input from the console is also done using the Console class, but we need to parse the input into the proper type.

Example:

```
ring input = Console.ReadLine();
 x = int.Parse(input);
```

STATEMENTS

Statements direct the operation of the program.

Statements end with a semicolon (;), but blocks of code between curly braces ({}) simply end with the right curly brace.

It is best practice to use indentation and spacing to align statements and blocks of code. This makes your code much easier to read.

COMMENTS

Comments are used to document what the code does. This is useful not only for other programmers who may need to maintain your code, but future you as well!

A single-line comment begins with two slashes (//).

A block comment starts with a slash and an asterisk (/*) and ends with the same two symbols in reverse (*/).

IDENTIFIERS

Any name created in C# is called an identifier. Identifiers may be used to name classes, methods, variables, and so on.

A keyword is one that is reserved by the language. It may not be used as an identifier.

NAMING AN IDENTIFIER

Start each identifier with a letter or underscore. Use letters, underscores, or digits for subsequent characters.

Don’t use keywords!

DECLARING AND INITIALIZING VARIABLES

We use variables to store data. For each variable we use, we must declare it (specifying its data type):

```
pe variableName;
```

In order to initialize a variable, we have to assign it a value:

```
riableName = value;
```

CONSTANTS

A constant stores a value that cannot change as the program executes.

Declare a constant by preceding the normal initialization with the keyword const and capitalizing all letters in the name of the variable.

DATA TYPES

VALUE TYPES:

- Variables have a value
- Space allocated on stack
- Examples: int, double, float.

DATA TYPES

REFERENCE TYPES:

- Variable is just a reference allocated on the stack
- Reference is a “type-safe” pointer
- Data space allocated on heap
- Examples: string, arrays.

DATA TYPES

C# Type Alias	CLS Type	Size (bits)	Suffix	Description	Range
sbyte	SByte	8		signed byte	–128 to 127
byte	Byte	8		unsigned byte	0 to 255
short	Int16	16		short integer	-32,768 to 32,767
ushort	UInt16	16		unsigned short integer	0 to 65535
int	Int32	32		integer	-2,147,483,648 to 2,147,483,647
uint	UInt32	32	u	unsigned integer	0 to 4,294,967,295
long	Int64	64	L	long integer	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	UInt64	64		unsigned long integer	0 to 18,446,744,073,709,551,615
char	Char	16		Unicode character	any valid character (e.g., 'a' , '*', '\x0058' [hexadecimal], or '\u0058' [Unicode])
float	Single	32	F	floating-point number	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
double	Double	64	d	double floating-point number	Range $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$
bool	Boolean	1		logical true/false value	true/false
decimal	Decimal	128	m	used for financial and monetary calculations	from approximately 1.0×10^{-28} to 7.9×10^{28} with 28 to 29 significant digits
bool	Boolean	true or false		used to represent true or false values	

ARITHMETIC EXPRESSIONS

ORDER OF PRECEDENCE

- Increment and decrement
- Positive and negative
- Multiplication, division, and remainder
- Addition and subtraction

ARITHMETIC OPERATORS

Operator	Name
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+	Positive sign
-	Negative sign

ASSIGNMENT STATEMENTS

An assignment statement consists of a variable, an equals sign, and an expression. When the assignment statement is executed, the value of the expression is determined and the result is stored in the variable.

ASSIGNMENT OPERATORS

Operator	Name
=	Assignment
+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division
%=	Modulus

CASTING

IMPLICIT CASTING

Assigning a less precise data type to a more precise data type will cause C# to automatically convert the less precise data type to the more precise data type. This is also called a widening conversion.

CASTING

EXPLICIT CASTING

We can code an assignment statement that assigns a more precise data type to a less precise data type. In this case, we must use parentheses to specify the less precise data type. This is also called a narrowing conversion.

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT:

- C# Importance and History
- How C# is compared to other programming languages.
- Types of applications developed using C#.
- Know common C# IDEs
- How to write a Hello World program using C#.
- Data types, statements, and variables in C#.

CONTROL STATEMENTS AND LOOPS

BOOLEAN EXPRESSIONS

BOOLEAN EXPRESSIONS AND CONTROL STATEMENTS

Recall that a boolean expression is one that evaluates to either true or false. We can use these expressions to determine the flow of our control statements.

RELATIONAL OPERATORS

The six relational operators (listed in following slide) are used to compare operands which are primitive data types.

Operands may be literals, variables, arithmetic expressions, or keywords.

RELATIONAL OPERATORS

Operator	Name
==	Equality
!=	Inequality
>	Greater Than
<	Less Than
>=	Greater Than or Equal
<=	Less Than or Equal

EXAMPLES ON BOOLEAN EXPRESSIONS

```
cent == 2.3 // equal to a numeric literal
 y' // equal to a char literal
 false // equal to the false value
 0 // not equal to a numeric literal
/ greater than a numeric literal
 // less than a variable
 500 // greater than or equal to a numeric literal
 reorderPoint // less than or equal to a variable
 isValid is equal to true
 isValid is equal to false
```

LOGICAL OPERATORS

&& And

|| Or

! Not

SELECTION STATEMENTS

- If Statement
- Switch Statement

THE IF STATEMENT

An if statement identifies which statement to run based on the value of a Boolean expression.
Example:

```
:ion = true;
.on)

 System.out.println("Variable is set to true.");


 System.out.println("Variable is set to false.");
```

THE SWITCH STATEMENT

The switch statement is a control statement that selects a switch section to execute from a list of candidates.

Example:

```
tch = 1;
eSwitch)
  System.out.println("Case 1");

  System.out.println("Case 2");

: System.out.println("Default case");
```

LOOPS

- The while loop
- The do loop
- The for loop
- The for each loop

THE WHILE LOOP

A while loop statement repeatedly executes a target statement as long as a given condition is true.

```
while(Boolean_expression)
{
        //Statements
}
```

THE DO LOOP

A do loop is similar to a while loop, except that a do loop is guaranteed to execute at least one time.

```
do
{
        //Statements
}while(Boolean_expression);
```

THE FOR LOOP

A for loop allows you to write a loop that needs to execute a specific number of times. A for loop is useful when you know how many times a task is to be repeated.

```
for(initialization; Boolean_expression; update)
{
   //Statements
}
```

THE FOREACH LOOP

This is mainly used to iterate through a collection of elements inside collections, such as arrays.

```
foreach(declaration in expression)
{
        //Statements
}
```

FOREACH EXAMPLE

```
umbers = {10, 20, 30, 40, 50};

.nt x in numbers ){
.e.WriteLine( x );
.e.WriteLine(",");

/riteLine("\n");
  names ={"James", "Larry", "Tom", "Lacy"};
.ng name in names ) {
.e.WriteLine( name );
.e.WriteLine(",");}
```

SIMPLE CONTROL STATEMENTS

COMPARE NUMERIC VARIABLES

We can use the six relational operators to compare numeric variables. These allow us to write Boolean expressions (which evaluate to either true or false).

CONTROL STATEMENTS

BREAK STATEMENT

The break statement is used to exit the current loop. In the case of multiple nested loops, a labeled break statement may be used to differentiate.

CONTROL STATEMENTS

CONTINUE STATEMENT

The continue statement is used to skip any remaining statements in the current loop and jump to the top of the current loop. A labeled continue statement may be used to jump to the top of a labeled loop.

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT:

- How to evaluate Boolean expressions.
- Know how to use logical and relational operators.
- Use if and switch statements
- Use for, while, do while, and enhanced for loop.
- Comparing numeric variables.
- Comparing string variables.
- How to use continue and break statements.

METHODS

WHAT IS A METHOD?

A method is a sequence of instructions that performs a specific task, and those instructions are packaged as a single unit.

DESCRIPTION

- To allow other classes to access a method, use the public keyword. To prevent other classes from accessing it, use the private keyword.
- Methods that don't return any data should have the void keyword as the return type.

DESCRIPTION

- Names of methods should typically start with verbs. Two very commonly used examples of this are methods which either set or return the values of instance variables, called setters and getters respectively.

SYNTAX

```
ırnType methodName([parameterList])

ents of the method
```

EXAMPLE

```
.ng getCode()

code="Java 101";
code;
```

METHOD OVERLOADING

When you create two or more methods with the same name but with different:

- Parameter type
- Parameter order
- Parameter count
- Passing method (in, out, ref)

CALLING METHODS

- When calling a method with no arguments, we include an empty set of parentheses after the method name.
- If a method returns a value, we can code an assignment statement to assign the return value to a variable. The data type of that variable must be compatible with the data type of the return value.

SYNTAX

```
methodName(argumentList)
```

EXAMPLES

```
.ntToConsole();

:Code(productCode);

:e = product.getPrice();
```

PASSING PARAMETERS TO METHODS

There are three ways to send parameters to methods in C#

- Pass by value
- Pass by reference
- Pass by out

RECURSION

- Recursion is a concept that aims to solve a problem by dividing it into smaller chunks in what is called divide and conquer.
- The idea is that a method can call itself, but with parameters that represent a smaller instance of the problem.
- You need to specify a stop condition to end the recursion.

EXAMPLE!

- Do the Factorial example.

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT:

- What are methods and why are they important
- Syntax for writing methods
- How to do method overloading
- How to call methods
- Passing parameters to methods
- Doing recursion

STRINGS

STRING VARIABLES

- A string consists of letters, numbers, and special characters strung together.
- Use "String" or "string" to declare a string variable.

STRING VARIABLES

- Strings are immutable!
- This means that once they are created, they cannot be changed!

CREATING STRINGS

Examples

```
String word ="Hello!";
char[] helloArray = { 'H', 'e', 'l', 'l', 'o'};
String helloWord = new String(helloArray);
```

JOINING STRINGS

- A string may be joined with another string by using the plus symbol (+). However, this will convert any other data type to a string.
- Another way is to use the string.Concat method, which takes two strings as parameters, and returns a new joined string.

COMPARING STRINGS

- You can compare strings in C# using ==, String.Compare, or using the Equals method of the string object
- Equals method can be used to ignore the case when comparing two strings

COMPARING STRINGS

Examples

```
// return true if the firstname is equal to Frank
firstName.Equals("Frank")
// return 0 if the two strings are equal(ignore case)
String.Compare(firstname, "Frank", true)
// equal to an empty string
firstName.Equals("")
// not equal to a string literal
!lastName.Equals("Jones")
// not equal to a null value
firstName != null
```

STRING FUNCTIONS

Examples

- int IndexOf(String str): Returns the index of the first occurrence of a certain substring. If the substring is not found, the function returns -1
- int LastIndexOf(String str): Returns the index of the rightmost occurrence of the certain substring.
- bool EndsWith(String suffix): Checks if the string ends with the a certain suffix.

STRING FUNCTIONS

Examples

- String Replace(char oldChar, char newChar): Returns a copy of the string that has oldChar replaced with newChar.
- String[] Split(separator(s)): Splits the string around matches of given char separator(s), and returns the words as an array of strings.

STRING FUNCTIONS

Examples

- String substring(int beginIndex): Returns a new a substring that starts at a specified index
- String ToUpper(): Returns a string that has all upper case chars.
- String trim(): Omits leading and trailing whitespaces.

STRINGBUILDER

- Strings can leave many unused objects in the memory when you do a lot of operations on them, as a new copy is made after each operation.
- It is better to use StringBuilder when you do a lot of string operations.
- Unlike Strings, objects of type StringBuilder are mutable, so they can be modified.

STRINGBUILDER EXAMPLE

```
ingBuilder strBuff = new StringBuilder("test!");

Buff.Append("\t Super!");

sole.WriteLine(strBuff);
```

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT:

- What are strings
- How to define and initialize strings
- Joining strings
- Comparing strings
- String functions
- Mutable strings (StringBuilder)

EXCEPTIONS AND ERROR HANDLING

TYPES OF ERRORS

- Syntax errors violate the rules for how C# statements must be written.
- Runtime errors don't violate syntax rules, but cause exceptions to be thrown which stop the execution of the application.
- Logic errors don't cause syntax or runtime errors, but produce incorrect results.

COMMON SYNTAX ERRORS

- Misspelling keywords
- Forgetting to declare a data type for a variable
- Forgetting an opening or closing parenthesis, bracket, quotation mark, or comment character
- Forgetting to code a semicolon at the end of a statement

HANDLING EXCEPTIONS

How Exceptions Work

An exception is thrown when the application can't perform an operation. An exception is an object created from one of several different Exception classes.

HANDLING EXCEPTIONS

How Exceptions Work

An application is said to catch any thrown exceptions and handle them, which may involve simply informing the user they need to provide valid input or more complicated action.

HANDLING EXCEPTIONS

In order to catch an exception, we use blocks of statements, called the try statement (which contains code that may throw an exception) and exception handler (which details the appropriate response to an exception).

SYNTAX

```
statements

ch (ExceptionClass exceptionName)

statements
```

EXAMPLE

```
louble subtotal = 0.0;
    try{
    Console.WriteLine("Enter subtotal:");
    subtotal = Double.Parse(Console.ReadLine());
    }
    catch (FormatException e)
    {
        Console.WriteLine("Error! Invalid number.");
    }
```

HAVING MULTIPLE CATCHES

```
try{ // statements causing exception

catch( ExceptionName e1 )
    // error handling code

catch( ExceptionName e2 )
    // error handling code

catch( ExceptionName e3 )
    // error handling code
```

THROW STATEMENT

Gives the programmer the ability to generate an exception.

```
r new Exception("Interst rate must be > 0.");
```

VALIDATING DATA

Preventing Exceptions

We will generally wish to prevent exceptions being thrown due to invalid data. We can use data validation to display an error message when the user inputs invalid content until all entries are valid.

NUMERIC ENTRIES

- Make sure the entry has a valid numeric format
- Make sure that the entry is within a valid range. This is known as range checking.

VALIDATING STRING INPUT

Use the TryParse!

```
price;
Double = Double.TryParse(inputString, out price);
uble) {
ess the input here
```

SOFTWARE TESTING

To test an application is to run it to make sure it works correctly, trying every possible combination of input data and user actions to be certain it work in every case.

SOFTWARE TESTING PHASES

- Check the user interface to make sure that it works correctly.
- Test the application with valid input data to make sure the results are correct.
- Test the application with invalid data or unexpected user actions. Try everything you can think of to make the application fail.

SOFTWARE DEBUGGING

Debugging an application means actually fixing the errors (or bugs) identified by testing the it. The app must then be tested again to make sure the fix hasn't caused more errors.

LOGICAL ERRORS

Logical errors are errors that will not make your program run as expected.

EXAMPLES:

- Initializing variables to the wrong value.
- Doing the wrong operation (doing addition instead of a difference).

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT

- Error Types.
- How to handle exceptions and runtime error.
- How to validate input.
- How to test and debug your code.
- How to fix syntax errors.
- How to detect logical errors.

DOCUMENTA-TION IN C#

DOCUMENTATION IN C#

C# provides support for generated documentation.

This documentation can be generated using XML tags.

DOCUMENTATION IN C#

To generate documentation, the programmer needs to use the triple back-slash.

Then, you can use Visual Studio to generate the documentation from those comments.

DOCUMENTATION IN C#

DEMO!

CODE BEST PRACTICES

TOPICS

- Coding best practices
- Code refactoring

BEST PRACTICES

CLEAN CODE

A lot of time is wasted due to poorly written and organized code. If we take the time to clean up our code from the beginning, we can avoid negative consequences later on.

BEST PRACTICES

STRATEGIES FOR WRITING CLEAN CODE

- Give things meaningful names
- Write comments only when necessary (make the code so clear it speaks for itself)
- Create small functions
- Follow a specific formatting structure

BEST PRACTICES

CODE REVIEW

One way to evaluate the quality of a feature we've coded is to submit it for *code review*. This is a systematic examination of the software.

BEST PRACTICES

TYPES OF CODE REVIEW

- Over-the-shoulder
- Email pass-around
- Pair programming
- Tool-assisted

BEST PRACTICES

REFACTORING CODE

When we *refactor* our code, we go back and restructure it without changing its behavior. As far as the user's interaction with the application is concerned, nothing changes.

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT

- How a clean code looks like.
- How to write clean code.
- How to do code reviews.
- The types of code review.
- How to refactor code.

ENUMERATI-ONS

WHAT IS AN ENUMERATION?

An enumeration contains a set of related constants. The constants are defined with the int type and assigned values from 0 to the number of constants in the enumeration minus 1.

WHAT IS AN ENUMERATION?

Enumerations are type-safe. This means we cannot specify another type where an enumeration type is expected.

ENUMERATION SYNTAX

```
ı EnumerationName

ʌME1 [,

ʌME2]…
```

ENUMERATIONS

DEMO!

ARRAYS

WHAT IS AN ARRAY?

An array is an object that contains one or more items called elements. All elements of an array must be of the same type.

Arrays have a fixed length or size that indicates the number of elements that it contains. You can code the number of elements as a literal, a constant, or a variable of type int.

CREATING ARRAYS

To create an array you must declare a variable of the correct type and instantiate an array object that the variable refers to.

The elements of an array are referred to by their index. Indexing begins with 0, so an index of 0 refers to the first element; an index of 3 refers to the fourth element.

INITIAL VALUES IN ARRAYS

Depending on the data type for the array, each element will be given an initial value.

- 0 for numeric arrays
- false for booleans
- '\0' (zero) for characters
- null for objects

DECLARING AND INSTANTIATING ARRAYS

```
ayName = new type[length];
```

Or

```
ayName;
 new type[length];
```

DECLARING AND INSTANTIATING ARRAYS

```
itles = new String[3];

rices;
w double[4];
```

DECLARING AND INSTANTIATING ARRAYS

```
itles = new String[3];//Array of Strings


ITLE_COUNT = 100;// Code that uses a constant
itles = new String[TITLE_COUNT];
```

DECLARING AND INSTANTIATING ARRAYS

```
: uses a variable
it = Console.ReadLine();
)unt = int.Parse(input);
:itles = new String[titleCount];
```

ACCESSING ARRAY ELEMENTS

```
.ndex];
  // Refer to the third element
```

ASSIGNING VALUES TO ARRAYS

```
ayName = {value1, value2, value3, …};


 new type[length];
] = value1;
] = value2;
] = value3;
```

USING FOR LOOP WITH ARRAYS

For loops are commonly used to process the elements in an array one at a time by incrementing an index variable. You can use the length field of an array to determined how many elements are defined for the array.

USING FOR LOOP WITH ARRAYS

SYNTAX

```
= 0; i < arrayName.length; i++)

:s
```

USING FOR LOOP WITH ARRAYS

Code that prints an array of prices to the console

```
rices = {14.95, 12.95, 11.95, 9.95};
 = 0; i < prices.length; i++)

teLine(prices[i]);
```

USING FOR LOOP WITH ARRAYS

Code that computes the average of the array of prices

```
= 0.0;
= 0; i < prices.length; i++)

es[i];

age = sum / prices.length;
```

USING FOREACH

The foreach loop lets you process each element of an array without the need to use indexes. The foreach loop simplifies the code required to loop through arrays.

FOREACH SYNTAX

```
ype variableName in arrayName)

ariableName is access elements
```

USING FOREACH

Code that prints an array of prices to the console

```
rices = {14.95, 12.95, 11.95, 9.95};
t element in prices)

teLine(element);
```

USING FOREACH

Code that computes the average of the array of prices

```
 = 0.0;
.ement in prices)

ıent;

:age = sum / prices.length;
```

THE ARRAY CLASS

The Array class contains several static methods that you can use to compare, sort, and search arrays.

Examples: Sort, BinarySearch, Clear.

REFERENCE AND COPY ARRAYS

REFERENCE AN ARRAY

You can create a reference to an array by assigning an array variable to an existing array. Both variables will refer to the same array, thus changes to one will be reflected in the other.

REFERENCE AND COPY ARRAYS

COPY AN ARRAY

The easiest way to copy an array is to use the CopyTo method of the array object. When you copy an array the new array must be of the same type as the source array.

HOW TO COPY ARRAYS

```
rades = {92.3, 88.0, 95.2, 90.5};
ercentages=new double [grades.Length];
To(percentages, 0);// 0 is where the copy starts
[1] = 70.2;
teLine("grades[1]=" + grades[1]);
```

TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays use two indexes to store data. Each element in the array is at the intersection of a row and column.

RECTANGULAR ARRAYS

The simplest type of two-dimensional array is a rectangular array, in which each row has the same number of columns.

RECTANGULAR ARRAYS

SYNTAX AND EXAMPLE

```
ayName = new type [RowCount,ColumnCount];
ers = new int[3,2]; // Array with 3 rows and 2 columns
```

JAGGED ARRAYS

A jagged array is a two-dimensional array in which the rows contain unequal numbers of columns. When you instantiate a jagged array, you specify the number rows, but not the number of columns.

JAGGED ARRAYS

SYNTAX

```
 arrayName = new type[rowCount] [ ];
need to create each row
```

JAGGED ARRAYS

EXAMPLE

```
local jagged array with 3 rows.
= new int[2][];
 int[2]; // Create a new array for row 0
 int[5]; // Create a new array for row 0
```

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT:

- What is an array.
- How to declare and instantiate an array.
- How to access and use array elements.
- How to use loops with arrays.
- How to use the Array class.
- How to copy arrays.
- How to create and use multi-dimensional arrays.

COLLECTIONS

WHAT ARE COLLECTIONS?

- Similar to an array, a collection is used to hold other objects.
- They are also more flexible and efficient than arrays.

COLLECTION TYPES

ARRAYLIST

An array list is a collection that's similar to an array, but it can change its capacity as elements are added or removed.

ARRAYLIST

EXAMPLE

```
ıyList will store elements as objects
ıumbers = new ArrayList();
ł(5);
:each (int i in numbers)

ısole.WriteLine(i);
```

ARRAYLIST METHODS

- add(object): Adds an object to the end of the list.
- Count: Returns the number of elements in the list
- Insert(index, object): Adds an object at a specific location.

HASHTABLE

- A Hashtable where elements are organized based on Key-Value pairs.
- Keys are used to access elements or values.

ARRAYLIST

EXAMPLE

```
ıt = new Hashtable();

", "John");
", "Paul");
.teLine(ht["001"]);
```

OTHER COLLECTIONS

- A stack is a LIFO (Last in First Out) data structure.
- A Queue is a FIFO (First in First Out) data structure.

GENERIC COLLECTIONS

- Generics allows us to create typed collections, which can hold objects of any type.
- To declare a variable that refers to a typed collection, we need to list the type in angle brackets (<>) following the name of the collection class.
- To include, use "System.Collections.Generic";

GENERIC COLLECTIONS

EXAMPLE

```
:s = new List();
l(5);
:each (int i in numbers)

nsole.WriteLine(i);
```

RECAP

WHAT YOU SHOULD KNOW AT THIS POINT:

- Know what are collections.
- Know classes and namespaces that define collections.
- Generics and their role in collections
- Define and use Arraylists and Lists.
- Define and use Hashtables. Know when to use Hashtables.