**Programming Fundamentals**

**What is a Computer Program?**

A sequence of instructions stored in the computers memory that a computer follows in order to perform a task.

- A statement is a complete instruction that causes the computer to perform some action.

**Why Program?**

**CONCEPT:** Computer programs can do many different jobs because they are programmable.

To automate a specific task

!.

#.

To solve a given computing problem

**Programming Is Both An Art and a Science** Common things that must be designed for programs:

The logical flow of instructions

●

The mathematical procedures

●

The layout of the programming statements

●

The appearance of the screen

●

The way information is presented to the user

●

The programʼs “User Friendliness”

●

Manuals, help systems, and/or other forms of written documentation ●

**Computer Systems: Hardware and Software**

**CONCEPT:** All computer systems consist of similar hardware devices and software components

Hardware

●

–

The central processing unit (CPU)

–

Main Memory (Random Access Memory or RAM) –

Secondary Storage (Disk Drive, Solid State Drive, USB Drive) –

Input Devices (Keyboard, Mouse)

–

Output Devices (Monitor, Printer)

Software

●

–

Operating Systems (MacOS, Windows 10, Android, iOS) –

Application Software

**The Central Processing Unit**

The CPUʼs job is to fetch instructions, follow instructions, and produce some

●

result data

CPU consists of two parts:

●

–

Control Unit

–

Arithmetic and Logic Unit (ALU)

When running a program, the CPU is engaged in a process known as fetch/

●

decode/execute cycle

Can only understand binary language - a sequence of 0ʼs and 1ʼs ●

**Programming Languages**

**CONCEPT:** A program is a set of instructions that a computer follows in order to perform a task. A programming language is a special language used to write computer programs.

There are two types of high level computer languages, compiled, and interpreter based. Compiled must be compiled beforehand. Interpreter based is compiled at runtime.

**What is a Program Made Of?**

**CONCEPT:** There are certain elements that are common to all programming languages.

Key Words (Reserved Words, Key Words in C# are lowercase.)

●

Operators (=,+,-,\*)

●

Punctuation (semi-colon, double quotes)

●

Programmer-Defined Names

●

Syntax (How to use the key words and operators in the programming language

●

youʼre using.)

**Sample Code Payroll.cs**

using System;

public class Payroll

{

public static void Main(String[] args)

{

int hours = 40;

double grossPay, payRate = 25.0;

grossPay = hours \* payRate;

System.Console.WriteLine("Your gross pay is $" + grossPay); }

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **What Tools Do You Need to Build A Computer Program?**

Programming Language

!.

–

A set of keywords and a special syntax for organizing program instructions. Source Code Editor

#.

–

For writing and editing code

Compiler

p.

–

Translate source code into a form that the computer understands (Machine Language)

–

Can help you notice syntax errors in your code

Debugger

q.

–

Allows you to step through a program and watch the value of variables change as the program executes

**Compiling and Running a C# Program**

Run the “Developer Command Prompt for CS 2017” and use the csc command

●

to compile a C# source file into a program, for example: csc Payroll.cs To executable a program type the executable file name, for example:

●

Payroll.exe

**Integrated Development Environment (IDE)**

An Integrated Development Environment (IDE) such as Visual Studio includes

●

all the tools you need to write, build, and run computer programs. IDE consists of a text editor, compiler, debugger, and other utilities integrated

●

into package with a single set of menus

**The Programming Process**

**CONCEPT:** The programming process consists of several steps, which include design, creation, testing, and debugging activities.

!.

Clearly defined what the program is to do (example follows) –

Purpose: To calculate the employees weekly gross pay –

Input: hoursWorked, payRate

–

Process: grossPay = hoursWorked \* payRate

–

Output: The value of the grossPay variable

Visualize the programming running on a computer (example follows)

#.

–

Enter hours worked:\_\_\_\_

–

Enter hourly pay rate:\_\_\_\_

–

Your gross pay is:\_\_\_\_

p.

Use design tools to create a model of the program –

Pseudocode

// Display “Enter the hours worked”

// Input hoursWorked

// Display “Enter the hourly pay rate”

// Input payRate

// Multiply hoursWorked by payRate and store result in grossPay variable // (grossPay = hoursWorked \* payRate)

// Display the value of the grossPay variable

–

Flowchart

q.

Check the model for logical errors

v.

Enter the code and compile it

w.

Correct any errors found during compilation. Repeat steps 5-6 as many times as necessary.

x.

Run the program with test data for input

y.

Correct any runtime errors while running the program. Repeat steps 5-8 as many times as necessary.

z.

Validate the results of the program.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

//Display "Enter the customer's maximum amount of credit" //Input maxCredit

//Display "Enter the amount of credit used by the customer" //Input usedCredit

//Subtract usedCredit from maxCredit

//(availCredit = usedCredit - maxCredit)

//Display the value of the availCredit variable

**Step 1:**

Purpose: To calculate the current account balance

Input: startBal, depBal, withBal, intRate

Process:

currentBalance = startBalance + depositAmount - withdrawAmount interestAmount = currentBalance \* interestRate

currentBalance = currentBalance + interestAmount

Output: Display the curBal value

**Step 2:**

Enter the starting balance: 1000

Enter the amount to deposit: 200

Enter the amount to withdraw: 700

Enter the monthly interest rate (as a percent of 1.0): 0.01

Your current account balance is: 505.00

**Step 3:**

//Display "Enter the starting balance"

//Input startBal

//Display "Enter Total Dollar Amount of Deposits Made"

//Input depBal

//Display "Enter Total Dollar Amount of Withdrawls Made"

//Input withBal

//Display "Enter Monthly Interest Rate (Out of 1.0)“

//Input intRate

//intAmount = (((startBal + depBal) - withBal) \* intRate)

//Display “Your interest amount is” + intAmount

//curBal = (intAmount + startBal + depBal - withBal)

//Display “Your current balance is” + curBal

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Special Characters (Punctuation)**

**Characters Name Meaning**

( ) Opening & Used to indicate that you are using a method.

Closing Parentheses

{ } Opening & Used to enclose a group of statements,

Closing Braces such as the contents of a

method or a class.

“ “ Double Quotation This encloses a string of characters.

; Semi-colon Marks the end of a complete programming statement.

// Double Slash Marks the beginning of a single line comment.

‘ ‘ Single Quotation Encloses a single character. /\* \*/ Slash Asterisk Encloses a block of comments. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**C# Syntax Rules**

–

C# is a case-sensitive programming language.

–

All C# programs must be stored in a file with a name that ends with .cs –

Comments are ignored by the compiler.

–

Every C# program must have a method named Main as an entry point. –

For every left brace, or opening brace, there must be a corresponding right brace or closing brace.

–

Statements are terminated with a semi-colon.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Console.Write keeps adding to the same line unless specified.

Console.WriteLine adds a new line every time.

**Variables and Types and Literals**

“Variables” are simply storage locations for data. You can place data into them and retrieve their contents as part of a C# expression. The interpretation of the data in a variable is controlled through “Types”.

A literal is a value that is written into the code of the program.

C# is a “Strongly Typed” language. Thus all operations on variables are performed with consideration of what the variableʼs “Type” is. There are rules that define what operations are legal in order to matin the integrity of the data you put in a variable.

If you need to declare multiple variables of the same data type you can separate them by commas.

**The Boolean Type**

Boolean types are declared using the keyword, bool.

●

They have two values: true or false.

●

They take up 1 byte (i.e. 8 bits) of storage.

●

Example:

●

bool switch = true;

**Integral Types**

In C#, an integral is a category of types.

●

They are whole numbers, either signed (e.g. -5) or unsigned (e.g. 5). ●

**Floating Point Types**

A C# floating point type is either a float, double, or decimal and represents a

●

real number with a decimal point (e.g. 3.14).

For financial calculations, the decimal type is the best choice because you can

●

avoid rounding errors.

**Char and String**

The char type is a 2 byte Unicode character enclosed in single quotes (e.g.

●

char letter = ‘Aʼ;) as defined by the Unicode Standard.

A string is a sequence of text characters. You typically create a string with a

●

string literal, enclosed in double quotes (e.g. string sentence = “This is an example of a string.”;)

String concentation is the connection of two strings using + ●

**Assignment Operator**

int seniorAge; //not assigned a value

int adultAge = 18; //assigned a value

double pi = 3.1415926535 d; //suffix doubles values with d float circleRadius = 64.869f; //suffix float values with f

decimal acctBalance = 8765.43m; //suffix decimal values with m string userGreeting = “welcome to the game…”;

char currentLetter = ‘Rʼ;

**How to Write to the Screen**

**CONCEPT:** The Write and WriteLine methods are used to display text output. They are part of the .NET Framework, which is a collection on prewritten classes and methods for performing specific operations.

**Escape Sequence Characters**

\a Bell (alert)

\b Backspace

\f Formfeed

\n New line

\r Carriage return

\t Horizontal tab

\v Vertical tab

\' Single quotation mark

\" Double quotation mark

\\ Backslash

\? Literal question mark

\ ooo ASCII character in octal notation

\x hh ASCII character in hexadecimal notation

\x hhhh Unicode character in hexadecimal notation if this escape sequence is used in a wide-character constant or a Unicode string literal. For example, WCHAR f = L'\x4e00' or WCHAR b[] = L"The Chinese character for one is \x4e00".

**Identifier Naming Rules**

The first character must be a letter (a-z or A-Z), or a underscore (\_).

●

After the first character, you may use letters, digits, or underscore (\_)

●

Uppercase and lowercase characters are distinct. This means itemsOrdered is

●

not the same as items ordered.

Identifiers cannot include spaces.

●

**Integer Division**

When both operands of the vision operator are integers, the operator will

●

perform integer division.

The result of integer division is an integer, if there is a remainder, it will be

●

discarded.

Convert one of the operands to floating point number if you want a floating

●

point result.

**Precedence of Arithmetic Operators** Highest -

\* / %

Lowest +

**The Math Class**

The Math class contains methods that are useful for performing complex

●

mathematical operations.

Use the Math.Pow method to return a specified number raise to the specified

●

power

Use the Math.Sqrt method, to return the square root of a specific number

●

Use the Math.Round method, to round to the nearest integer or to the

●

specified number of fractional digits

**Conversion Between Simple Data Types**

**CONCEPT:** Before a value can be stored in a variable, the valueʼs data type must be compatible with the variableʼs data type.

Use **cast operator** to manually convert a value●

double doubleMark = 88.88;

int intMark = (int) doubleMark; // = 88

double doubleResult = (double) intMark / 10; // = 8.8

**Creating Named Constants with const**

**CONCEPT:** The **const** key word can be used in a variable declaration to make the variable constant. Named constants are initialed with a value, and that value cannot change during the execution of the program.

const double gestate = 0.05;

const int minAge = 18;

const int maxAge = 65;

**Scope**

**CONCEPT:** A variableʼs scope is the part of the program that has access to the variable.

**Reading Keyboard Input**

**CONCEPT:** The **Console.ReadLine** method reads the next line of characters from the keyboard. This stops the program and waits for the user to hit enter. If you want to store a number you need to parse it using **.Parse** (e.g. age = int.Parse(Console.ReadLine);)

**Common Errors to Avoid**

Mismatched braces, quotation marks, or parentheses

●

Misspelling key words

●

Using capital letters in key words

●

Using inconsistent spelling of variable names

●

Using inconsistent case of letters in variable names

●

Inserting a space into a variable name

●

Forgetting a semicolon at the end of the statement

●

Assigning a **double** literal to a **float** variable

●

Using commas or other currency symbols in numeric literals

●

Unintentionally performing integer division

●

Forgetting to group parts of a mathematical expression

●

Inserting a space in a combined assignment operator ●

**Relational Operators**

< Less Than

<= Less Than or Equal To > Greater Than

>= Greater Than or Equal To != Not Equal To

== Is Equal To

**Logical Operators**

Logical operators connect two or more relational expressions into one **OR** reverse the logic of an expression.

**Boolean Operators**

Operator Name Description

! Not Logical Negation

&& And Logical Conjunction

|| Or Logical Disjunction

^ Ex-Or Logical Exclusion

**Truth Table for Operator !**

p !p

True False

False True

**Example**

int age = 24, weight = 140;

!(age > 18) - False

!(weight == 150) - True

**Truth Table for Operator &&**

p1 p2 p1 && p2

false false false

false true false

true false false

true true true

**Example**

age = 24, weight = 140;

(age > 28) && (weight <= 140) - False

(age > 18) && (weight <= 140) - True

**Truth Table for Operator || and ^**

p1 p2 p1 || p2 p1 ^ p2

false false false false

false true true true

true false true true

true true true false

**Example**

age=24, weight=140;

(age > 34) || (weight >= 150) - False

(age > 18) || (weight < 140) - True

(age > 34) ^ (weight > 140) - False

(age > 34) ^ (weight >= 140) - True

**Conditional Operators (? and :)**

A conditional operator evaluates an expression based on a condition.

**C# Syntax**

boolean expression ? expression1 : expression2;

The symbol ? and : appearing together is called a conditional operator. (Also known as a ternary operator.)

**Example of Conditional Operators**

max = (num1 > num2) ? num1 : num2;

min (num1 < num2) ? num1 : num2;

^————————— Long way of doing this.

if (num1 > num2)

{

max = num1;

}

else

{

max = num2;

}

**The if Statement**

The if statement is used to decide whether or not a statement is ran. Can use strings for the boolean expression.

if(boolean expression)

{

//True statement actions

}

**The if-else Statement**

if(boolean expression)

{

//True statement actions

}

else

{

//False statement actions

}

**The Nested if Statement**

To test more than one condition, an if statement can be nested inside another if statement

if(boolean expression)

{

if(boolean expression)

{

//True statement actions

}

}

**C# Switch Statement**

A switch executes statements based on the value of a variable or an expression.

**C# Syntax**

switch(switch expression)

{

// case compares the value to the switch expression.

// value has to match the switch expression.

case value1:

// Statements

// break; to terminate or exit the current data structure.

break;

case value2:

// Statements

break;

case valueN:

// Statements

break;

// default is required if it doesnʼt match any of the cases.

// default requires a break; as well.

default:

// Statements

break;

}

**Increment and Decrement Operator**

The (++) increment and (—) decrement operator.

●

++ and — are operators that add and subtract one from their operands.

●

The ++ and — can go before (prefix mode) or after (postfix mode) of the

●

variable

In prefix mode value is incremented/decremented immediately.

○

In postfix mode value is incremented/decremented after executing the

○

current statement.

int number = 4;

Console.WriteLine(++number); //5 ————————————————— int number = 4;

Console.WriteLine(number++); //4

**Loops**

A loop is the part of a program that repeats. There are two main categories of loops:

!.

Counter-Controlled Loop

Use a counter to count the number of times loop has repeated.

○

Use when you know exactly how many times to repeat a loop. ○

#.

Event(Condition)-Controlled Loop

Loop until an event (condition) occurs. ○

**C# While Loop Statement Syntax**

while (boolean expression) {

//Statements

}

**Loop Design Strategies**

The key to designing a loop is to identify the code that needs to be repeated

●

and write a condition for terminating the loop.

**Step 1:** Identify the statements to be repeated.

○

**Step 2:** Wrap these statements in a loop as follows: ○

while (true)

◆

{

// Statements

}

**Step 3:** Code the loop-continuation-condition and add appropriate

○

statements for controlling loop.

while (loop-continuation-condition)

◆

{

// Statements

// Additional Statements for controlling the loop

}

**Looping Structures**

!.

The while loop (Pre-test Loop). The syntax is as follows:

while (loop-continuation-condition) {

// Loop Body

}

A do-while loop (Post-test Loop) is the same as while loop except it executes

#.

the loop body first then checks the loop condition. The syntax is as follows: do

{

// Loop

} while (loop-continuation-condition);

p.

A for loop (Pre-test loop) is ideal for performing a known number of iterations. The syntax is as follows:

for (initial-action; loop-continuation-condition; action-after-each-iteration) {

// Loop Body

}

for (int count = 1; count <= 100; count++)

{

Console.WriteLine(“I can C#”);

}

int count;

for (count = 1; count <= 100; count++)

{

Console.WriteLine(“I can C#”);

}

**(Same as while loop)**

int count = 1;

for (; count <= 100;)

{

Console.WriteLine(“I can C#”);

}

**Methods**

Methods can be used to define reusable code and organize and simplify coding.

A method definition consists of a method name, parameters, return value type, and body.

A value method returns a value.

A void method executes the code.

**modifier** - static

**returnValueType** - int (or void)

**MethodName** - Max

**(parameterList)** - (int num1, int num2)

{

}

**Example: Defining a Method**

static int Max(int num1, int num2)

{

int result;

if (num1 > num2)

{

result = num1;

}

else

{

result = num2;

}

return result;

}

**Example: Calling A Method**

Calling a method executes the code in the method:

There are two ways to call a method.

**If a method returns a value, a call to the method is treated as a value.**

**For Example:**

int larger = Max(3, 4);

Console.WriteLine( Max(3,4) );

String name = Console.ReadLine();

**If a method returns void, a call to the method must be a**

**statement.**

**For Example:**

Console.WriteLine(“Welcome to C#”); **Array**

**How to Create an Array**

There are some thing you need to know before you can create an array: ●

What type of data will be stored: each element of the array must be of the

○

same type of data

How many elements: normally once an array is created it cannot be made

○

to hold more elements than it was originally designed to hold.

Once you know about an array you can give it a name: the naming follows the

●

same naming conventions that we use for variables.

**C# Syntax Arrays**

**Declare Array:**

double[] myList;

**Create Array:**

myList = new double[arraysize]; myList = new double[10];

**Access Array Elements:** myList[index];

**Load Data Into An Array**

There are several ways to load data: ●

Declare the array with known data

○

User/Program entered data

○

From an external source such as a file ○

Except for declaring the array with known data the loading of data into an

●

array involves looping; this is why the size becomes important.

**Logical Size vs. Physical Size**

An array can be either completely full of data, or be only partially filled

●

When an array is declared, given both size and data type, the computer only

●

allocates a block of memory but does not give the elements any data; we have to do that manually or programmatically.

**Physical Size:** the physical, declared, size of the array.

●

**Logical Size:** the number of elements with actual data.

●

**Logical Size <= Physical Size** ●

**Manually Load Data**

We can declare, and initialize, an array with pre-defined, known, data as

●

shown below:

string[] dayOfWeek = {“SUN”, “MON”, “TUE”, “WED”, “THU”, “FRI”, “SAT”}; **Program Generated Array**

Load an array with consecutive integer values: ●

//Program generated array

int[] someNumbers = new int[10]; for (int index = 0; index < 10; index++) {

someNumbers[index] = index; }

**User Entered Data**

You can have the user enter the data (the example below assumes the

●

existence of the GetInteger() method:

// user entered data

int[] userEntered = new int[5];

for (int index = 0; index < 5; index++)

{

userEntered[index] = int.Parse(Console.ReadLine()); }

**Initializing Array with Input Numbers**

Console.WriteLine(“Enter values for array:”);

for(int index = 0; index < myList.Length; index++) {

myList[index] = double.Parse(Console.ReadLine()); }

**Initializing Array with Random Values**

Random keygen = new Random();

for(int index = 0; index < myList.Length; index++)

{

myList[index] = keygen.Next(0,101);

}

**Displaying Arrays**

foreach(double num in myList)

{

Console.Write(num + “\_”);

}

**OR**

for(int index = 0; index < myList.Length; index++)

{

Console.WriteLine(myList[index] + “\_”);

}

**Summing All Elements**

double sum = 0;

for(int index = 0; index < myList.Length; index++)

{

sum += myList[index];

}

**Finding the Largest Element (Linear/Sequential Search)**

double largest = myList[0];

for(int index = 1; index < myList.Length; index++)

{

if(myList[index] > largest)

{

largest = myList[index];

}

}

**Finding the Smallest Index of the Largest Element (Linear/Sequential Search)** double largest = myList[0];

int indexOfLargest = 0;

for(int index = 1; index < myList.Length; index++)

{

if(myList[index] > largest)

{

largest = myList[index];

indexOfLargest = index;

}

}

**Random Shuffling/Swapping Values**

Random keygen = new Random();

for(int index = 0; index < myList.Length; index++)

{

int randomIndex = keygen.Next(myList.Length);

int temp = myList[index];

myList[index] = myList[randomIndex];

myList[randomIndex] = temp;

}

**Shifting Elements**

double temp = myList[0];

for(int index = 1; index < myList.Length; index++)

{

myList[index-1] = myList[index];

myList[myList.Length-1] = temp;

}

**Simplify Coding**

string[] monthNames = {“JAN”, “FEB”, “MAR”, “APR”, “MAY”, “JUN”, “JUL”, “AUG”, “SEP”, “OCT”, “NOV”, “DEC”};

Console.Write(“Enter the month #(1-12):”);

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

Console.WriteLine(monthNames[monthNum - 1]);

**Bubble Sort**

**static void** bubbleSort(**int** []arr)

{

**int** n = arr.Length;

**for** (**int** i = 0; i < n - 1; i++)

**for** (**int** j = 0; j < n - i - 1; j++)

**if** (arr[j] > arr[j + 1])

         {

         // swap temp and arr[i]

**int** temp = arr[j];

         arr[j] = arr[j + 1];

         arr[j + 1] = temp;

  }

 }

**Selection Sort**

**static void** sort(**int** []arr)

    {

**int** n = arr.Length;

        // One by one move boundary of unsorted subarray         **for** (**int** i = 0; i < n - 1; i++)

        {

            // Find the minimum element in unsorted array             **int** min\_idx = i;

**for** (**int** j = i + 1; j < n; j++)

**if** (arr[j] < arr[min\_idx])

                    min\_idx = j;

            // Swap the found minimum element with the first             // element

**int** temp = arr[min\_idx];

            arr[min\_idx] = arr[i];

            arr[i] = temp;

        }

    }

**OR**

for (int scanIndex=0; scanIndex < array.Length - 1; scanIndex++) {

//TEMP// double currentSmallestValue = array[scanIndex]; int currentSmallestValueIndex = scanIndex;

for(int index = scanIndex + 1; index < array.Length; index++) {

if(array[index] < currentSmallestValue)

{

currentSmallestValue = array[index];

currentSmallestValueIndex = index;

}

}

if (currentSmallestValueIndex != scanIndex)

{

array[currentSmallestValueIndex] = array[scanIndex];

array[scanIndex] = currentSmallestValue;

}

}

**WITH PARALLEL ARRAYS**

static void SortByPoints(string[]playerNames, int[] playerPoints) {

for (int scanIndex = 0; scanIndex < playerPoints.Length - 1; scanIndex++) {

int currentSmallestValue = playerPoints[scanIndex]; int currentSmallestValueIndex = scanIndex;

for (int index = scanIndex + 1; index < playerPoints.Length; index++) {

if (playerPoints[index] < currentSmallestValue)

{

currentSmallestValue = playerPoints[index];

currentSmallestValueIndex = index;

}

}

if (currentSmallestValueIndex != scanIndex)

{

playerPoints[currentSmallestValueIndex] = playerPoints[scanIndex]; playerPoints[scanIndex] = currentSmallestValue;

string tempName = playerNames[currentSmallestValueIndex]; playerNames[currentSmallestValueIndex] = playerNames[scanIndex]; playerNames[scanIndex] = tempName;

}

}

}

**Defining Classes for Objects**

A class defines the properties and behaviour for objects ●

The state of an object is represented by data fields with their current

○

values.

The behaviour of an object is defined by methods.

○

Classes are definitions for objects and objects are created from classes.

Properties allows you to get (accessor) and/or set (mutator) a data field value.

●

Methods are used to perform some operation on the properties of the class. ●

●

(instance-level)

A method that does not use properties of the class is known as a static (class

●

level) method.

**Constructor**

A constructor is a specialize method with the same name as the class name

●

and no return type.

The purpose of a constructor is to initialize data fields to a known state.

●

The constructor that is executed only once during the life cycle of an object

●

and used by the new operator to create an object in the class. A constructor with no method parameters is known as a default or no

●

argument constructor.

A class can have more than one constructor with different method parameters

●

called overloaded constructors.

If a class includes constructors you MUST use one of those constructors with

●

the new operator (you cannot assume all classes include a default no argument constructor).

**Objects**

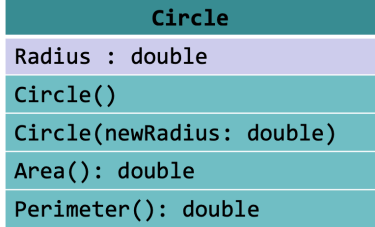
An object is an instance of the class created using the new operator

●

The syntax to create an object of a class is as follows: ●

**Syntax**

ClassName instanceName = new ClassName();



Circle

**Is the ClassName**

Radius : double

**Is the Properties/Data**

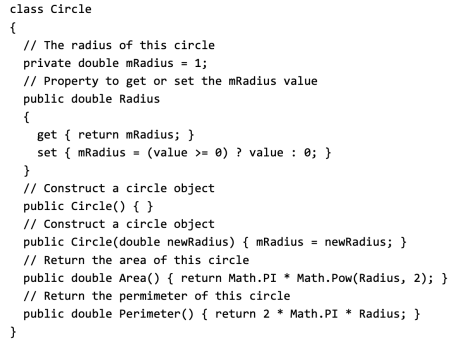
Circle()

Circle(newRadius: double)

Area(): double

Perimeter(): double

**Are all the methods of the class.**

****mRadius is the Member Property Name

Radius is the Property Name

get { return mRadius; } **get is used for reading values**

set { mRadius = (value >=0) ? value : 0; } **set is used for changing values** Requires both get and set to have a writeable value.

If you have only get or set it is read only.

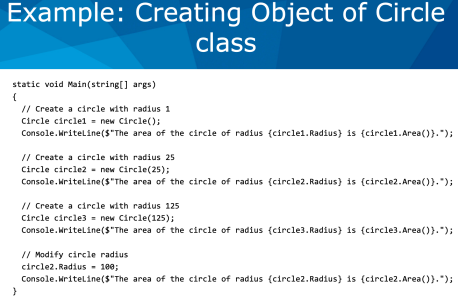
**Classes**

Classes contain:

!. #. p.

Data fields(Member Variables) Properties

Methods

Can get or set a property value

**or**

Call an object method

**or**

Modify a property value

**Visibility Modifier**

Visibility modifiers can be used to specify the visibility of a class and its members Visibility modifiers are used when you declare a class, a data field, a property, or a method (they are not to be used inside a method!)

**Specifier—Description**

public Visible from outside the class

protected Visible from the current class or derived class

private Visible only from the current class

none Visible within the same parent container (namespace, class, method) get is a accessor and set is a mutator.