**סיכום C#**

**תמיד בראש כל תוכנית יש לרשום (כדי לחסוך בכתיבה):**

using System;

static void Main()

{

Console.WriteLine(“Hello World!”);

} // Main

**פקודת הדפסה:**

Console.WriteLine("string....");

כאשר בתוך המחרוזת להדפסה יש {i} (כאשר i זה מס' גדול או שווה ל-0 ), הוא ידפיס בהתאם את הפרמטר ה-i-י שמוצב ברשימת הפרמטרים שבפקודה.

הערה – אם לא היה נרשם system, היה צריך להוסיף לפני system.

ב-C# ה- Null רשום כך null.

**פרמטרים ל-main:**

public static void Main(string[] args)

כאשר הפרמטר הראשון נמצא כבר באינדקס 0, שלא כמו ב-C.

שמתי לב שכאן הפונקציות מתחילות באות גדולה שלא כמו ב-JAVA.

**פקודת קריאה:**

Console.ReadKey();

string line = Console.ReadLine(); - If the CTRL+Z character is pressed when the method is reading input from the console, the method returns **null**

float x, y, z, u;

x = 12.67F;

מספר ממשי ללא ציון סוג נחשב ל-double ממשי 32 ביט, המהדר לא יקבל את הפקודה:

x = 12.67;

במידה ו-x מוגדר כ- float.

**משפט תנאי (למשל if):**

רק תנאי בוליאני שמחזיר true או false יכול לשמש כתנאי כניסה למשפט, כלומר ערך שונה מ-0 (כמו ב-c) הוא לא אמת ו-0 הוא לא שקר.

**מחרוזות:**

ניתן בין היתר לבצע השוואה בין 2 מחרוזות ע"י "==" (אך עדיף למשל השימוש ב- (s.Equals(str), וכמו כן ניתן לבצע שירשור של מחרוזות ע"י "+" .

string s="abcdefghijk";

string[] // tokenize strings

tokens = s.Split('.', ';'); // delimiters are . and ;

אי אפשר לכתוב לתוך המחרוזת כך (כלומר s[i] = c לא נתמך).

שיטות כמו Insert, Replace וכו' לא משנות את המחרוזת אלא יוצרות מחרוזות זמניות עם השינוי בתוכם.

בהחלט ניתן "לדרוס" מחרוזת במחרוזת גדולה יותר, במובן הזה מחרוזות מתנהגות כמו משתנים. מן הסתם הוא משחרר את המערך הקודם ויוצר חדש.

**שדות של מחלקה:**

תמיד נגדיר מהו סוג השדה (עבור כל משתנה של מחלקה), לרוב נגדיר כ-private.

**מערכים:**

הגדרת מערך (למשל):

int [] arr= new int[N];

int [] arr= new int[] {1,2,3};

מערך תמיד "יודע" מהו גודלו, ע"י arr. Length. כמו כן הוא תמיד מועבר לפי יחוס לפונקציה ולא לפי ערך כמו ב-c.

**מערכים רב מימדיים:**

int [,] arr= new int[N,M];

הערה: Length של מערך רב מימדי, זה מכפלת רוחב השורות ברוחב העמודות(כלומר M\*N).

הקריאה GetLength(i) כאשר i זה אינדקס, תחזיר את אורך אותו מימד (נגיד עבור 0 יחזיר N).

**לולאה**:

לולאת foreach(double d in arr).

**פונקציה:**

נגדיר כ-public (בד"כ).

**פרמטרים**:

משתני **מחלקה** **ומערכים** **תמיד** מועברים כמשתני יחס גם בלי שנציין דבר לפני כן.

משתנים **פרימיטיביים** [גם מערכים פרמטיבים למיטב הבנתי] מועברים לפי ערך (by value) כברירת המחדל, ניתן לשנות זאת ע"י המילים: ref,out,params, שזה אומר העבר לפי יחס (חייב לאתחל משתנה כזה לפני הקריאה), לכתיבה בלבד (כמו ערך מוחזר שמצפים לו מפונקציה ובו יהיה הערך בסופו (אסור לקרוא את הערך הזה בתוך הפונקציה הנקראת)) או מס' לא מוגדר של פרמטרים **בהתאמה**.

אחת מהמילים הללו תופיע גם בהכרזה של הפונקציה שאמורה לקבל את הפרמטר בצורה הנ"ל (לפני סוג המשתנה) וגם בקריאה לפונקציה (לפני שם המשתנה שמעבירים אליה).

משתנה שלא אותחל לא יכול להיות משתנה ref. נקבל הודעות שגיאה אם ננסה להשתמש ב-ref

שלא אותחל או אם מנסים לקרוא ממשתנה שמוגדר כ-out.

static void fonc(params double[] darr)

{ foreach(double v in darr){..} }

…

func(12, 23, 56, 11);//הקריאה

**בנאי**:

נגדיר כ-public ויהיה כשם המחלקה (לא מחזיר שום ערך).

**this**:

כהתייחסות למשתני המחלקה (בעת התנגשות של שמות בד"כ זה שימושי).

**קבועים**:

ע"י תוספת של המילה const, למשל: public const int N=2;.

משתנים לקריאה בלבד: להוסיף

משתנה שמוגדר כ-readonly למשל: public readonly int n;, הוא משתמש לקריאה בלבד, וניתן לבצע עליו השמה רק בבנאי.

const ידוע בזמן ריצה והוא כמו קבוע ולא ניתן לשנות את ערכו.

read only - ערכו ידוע בזמן ריצה ואפשר לעשות לו השמה רק בבנאי.

**מאפיינים (get/set):**

אם למשל יש לנו משתנה private int a; אז ניתן להגדיר עבורו set&get למשל כך:

public int A

{ get{ return a; } set {a=value;}}

כאשר value זה הערך שאותו מבקשים להשים במקומו של a.

הערה: יש לשים לב כי קריאה וכתיבה של a תעשה אך ורק ע"י השיטה A (אם זה נכתב לא מתוך מתודה השייכת למחלקה אליה שייך a( וזה כאשר a שדה פרטי)). לא חייב להגדיר set וגם get, ניתן גם רק אחד מהם.

Class c = new Class();

c.A = 5;//זה השמה במיין כאשר שם המחלק קלאס

**סדרנים:**

הקוד מקבל בין ה-[] כל פרמטר שרוצים , צריך שיהיה בה get כלומר אם נעשה השמה, וצריך להיות set אם נרצה לקחת את הערך, value היא מילה שמורה שזה ערך ההשמה ב-set.

ניתן לממש בעזרתו **מערך אסוציאטיבי**, לדוגמא אם נרצה לממש מחלקה של מטריצה, נרצה לפנות לאיבר בה כך: A[i,j] נכתוב:

public double **this**[int i, int j]

{

get { return arr[i, j]; }

set { arr[i, j] = value; }

}

כאשר מוגדר המערך arr כחלק ממשתני המחלקה.

private double[,] arr;

חשוב: הדרך בה קוראים "לסדרן", היא ע"י שם משתנה מסוג המחלקה בה מוגדר הסדרן, ו-[], כאשר בתוכם יש פרמטרים כלשהם שמועברים לסדרן.

למשל עבור הדוגמא שלעיל ונגיד שם המחלקה הוא matrix.

matrix A=new matrix();

..

A[4,5]; //קריאה לסדרן

**העמסת אופרטורים:**(הערך המוחזר מההעמסה לא יכול להיות void)

הרשימה של האופרטורים הניתנים להעמסה ב-C# :

(אם מעמיסים את != חייבים גם להעמיס את הנגדי שלו ==)

אופרטורים אונריים:

+ - ! ~ ++ -- true false

אופרטורים בינאריים:

+ - \* / % & | ^ <<>> == != > < >= <=

הערה: האופרטור [] אמנם לא מופיע ברשימה, אך הוא ניתן להעמסה ע"י הסדרן.

קוד של שיטה שהיא העמסת אופרטורים חייבת להיות מוגדרת public static תוך שימוש במילה השמורה operator.[חייב שלפחות משתנה אחד יהיה מוגדר ע"י המתכנת (לא פרימיטיבי)]

מבנה של שיטה של אופרטור אונרי:

**public static סוג ערך מוחזר operator סמל האופרטור(הגדרת פרמטר)**

מבנה של שיטה של אופרטור בינארי:

**public static סוג ערך מוחזר**

**operator סמל האופרטור(הגדרת פרמטר ימני, הגדרת פרמטר שמאלי)**

דוגמא להעמסת "+" שעושה חיבור של מטריצות:

public static Matrix operator +(Matrix A, Matrix B)

{

int i, j, n;

Matrix C;

n = A.N;

C = new Matrix(n);

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

C[i, j] = A[i, j] + B[i, j];

return C;

}

והקריאה עצמה תיראה כך:

Matrix a,b,c;

….

c=a+b;// כלומר נתנו משמעות חדשה **לחיבור** בין מטריצות מבחינת השפה

דוגמא להעמסה אונרית של "-":

public static Complex operator**-**(Complex right)

{

Complex c = new Complex();

c.realp = -right.realp;

c.imagp = -right.imagp;

return c;

}

**אופרטורי הסבה Casting**: שקול להעמסת = ו- ().

המתכנת יכול להגדיר אופרטורי הסבה משלו. אופרטורי הסבה יכולים להיות משתמעים (implicit) או מפורשים (explicit).

"הגדרת הפרמטר" היא למעשה לאיזה סוג פרמטר אנו מבקשים לעשות cast, והערך המוחזר, זה הוא הערך שאליו הוא יותמר.

הגדרת אופרטור הסבה משתמע: (נראה למשל כך: a=b;, b יותמר לסוג הערך של a).

public static implicit operatorסוג ערך מוחזר (הגדרת פרמטר)

הגדרת אופרטור הסבה מפורש:

public static explicit operatorסוג ערך מוחזר (הגדרת פרמטר)

לדוגמא אם נרצה לחשב את סכום כל איברי המטריצה:

double d=(double)M1;

…

public static explicit operator double(Matrix A)

{

int i, j, n;

double sum = 0;

n = A.N;

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

sum += A[i, j];

return sum;

}

**נציגים Delegates:**

נציגים Delegates הם המקבילים לפוינטר לפונקציה,

**הגדרה** (למעשה הגדרה של משתנה ייחוס, שהוא **סוג של משתנה** שכל פונקציה יכולה לקבל):

public delegate double func(double x);

סוג המשתנה שהגדרנו הוא למעשה נקרא: func.

כאשר רוצים להשתמש בסוג המשתנה (הדלגט) והוא הוגדר במחלקה Z, אז התייחסות לסוג המשתנה הנ"ל מחוץ למחלקה בה הוא קיים תיראה כך: Z.func

**אתחול**:

//עושים לו **ניו** ממש כמו למחלקה והפרמטר שלו הוא פונקציה שהכותרת שלה זהה לדלגט

func fd = new func(Functions.poly);//אין חובה מיד לעשות **ניו**

יש לשים לב כי poly זו פונקציה שמוגדרת כ-static במחלקת Functions.

ואז קריאה לפונקציה שנבחרה תיעשה כך(ממש דומה לפוינטר לפונקציה)

fd(5.5);

**event** – הודעה שאובייקט שולח לאובייקטים אחרים.

event הינו למעשה מקרה פרטי של delegate. הרעיון של event הינו להצביע על פונקציה והוא מספק מספר הגבלות אשר לא קיימות ב- delegate רגיל:

ניתן להגדיר event רק כחבר מחלקה (member) ולא ניתן בתוך פונקציה או כפרמטר לפונקציה

ניתן להפעיל event רק באותה מחלקה בה הוא נכתב

ניתן להוסיף ולהוריד פונקציות מ- event באמצעות =+, =- (כמו multicast delegate), אך לא ניתן לדרוס את מה שיש שם באמצעות =

הגבלות אלו מספקות ביטחון גבוה יותר, אשר מונע טעויות כמו דרישה של הפונקציות אליהם מצביע ה- event, ולכן אם לא נצטרך משהו שסותר הגדרות אלו (כגון פרמטר לפונקציה), נעדיף להשתמש ב- event.

בכדי להגדיר event צריך תחילה להגדיר delegate, לדוגמא:

 public delegate void SpeedDelegate(int newSpeed);

לאחר מכן יש להגדיר משתנה מסוג ה- delegate בצרוף המילה השמורה event:

public event SpeedDelegate SpeedChanged;

רישום פונקציה ל- event (הצבעה) והפעלת ה- event הינם בדיוק כמו ב- delegate (למעט ההגבלה של =).

//רישום הפונקציה לאירוע  
      c.SpeedChanged += new SpeedDelegate(PrintSpeed);

SpeedChanged(speed);//קריאה

**מרחב שמות:**

מגדירים בתוכו את המחלקה (סוגריים מסולסלות) ו-namespace somename

כדי להשתמש בו במקום כלשהו, עושים using somename.

using System;

namespace names1

{

דוגמא:

y = na.funs.sqrt(x);

עם שימוש ב-using na;

y = funs.sqrt(x);

**הורשה ב-:C#**

ניתן לרשת רק ממחלקה אחת.

הגדרה של הורשה:

Class שם מחלקה חדשה**:** שם מחלקת הבסיס

{

פרטי השדות נוספים / דורסים של המחלקה החדשה

}

**מילת המפתח base**:

משמש לקרוא לאחת הפונקציה הבונות (בנאי) של מחלקת האב, או לקרוא לאחד השיטות שלה ממחלקת הבן במקרה של התנגשות שמות (עבור הפונקציות).

הערה: אם לא התבצעה קריאה **בבנאי** של המחלקה הנגזרת לבנאי של מחלקת הבסיס, ייקרא כברירת מחדל **הבנאי ללא הפרמטרים** של מחלקת הבסיס.

לאחר ההורשה ניתן לקרוא לבנאי של האב למשל כך:

public point2():base() {..}

public point2(double xv, double yv):base(xv, yv) {…}

**גישה לשדות:**

מחלקות נגזרות יכולות לגשת לשדות public ו-protected של מחלקת **הבסיס** אבל לא לשדות ה-private שלו. שדות public של מחלקת הבסיס הם שדות public של המחלקה הנגזרת. שדות protected נגישות לשיטות של המחלקה הנגזרת, אבל לכל קוד אחר הן חסומות (כמו שדות private). משתנה internal הוא נגיש לשיטות ביחידת ההידור הנוכחי בלבד.

**דריסת שיטות:**

ניתן לדרוס במחלקה הנגזרת את השיטות המוגדרות במחלקת הבסיס ועדיין להשתמש בשיטות של מחלקת הבסיס לצורך המימוש. במצב זה רצוי להשתמש במילת המפתח new. אחרת הקומפיילר ייתן אזהרה.

**new** public void show()

{

**base.show();//** זאת קריאה לאותה פונקציה במחלקת הבסיס

Console.WriteLine("radius={0}, theta={1}", radius, theta);

}

הסבר: מה שאומר המימוש הנ"ל, זה אם נקרא ל- show של המחלקה היורשת, בהכרח הוא יופעל, ואם בכל זאת נירצה את המימוש הישן של הבסיס, נשתמש במילה base.

**מה פירוש הדבר "למה שהמשתנה היחס מצביע עליו בפועל"?**

לדוגמא הקוד הבא:

point c;

point2 c2;//מחלקה שיורשת מפוינט

c2=new point2();

c=c2;

בקוד הנ"ל המשתנה c הוא מסוג מחלקת הבסיס, אך בפועל מצביע למשתנה מסוג המחלקה הנגזרת. (אותו הדבר כאשר מגדירים מופע שנגזר מהבסיס, ומעבירים אותו לפונקציה שמצפה בפועל למשתנה מסוג הבסיס, זה מקביל למימוש הנ"ל)

**שיטות סטטיות:**

אם אנחנו רוצים שבמקרה של התנגשות עבור שיטה מסוימת השיטה שתורץ תהיה לפי **הגדרת המשתנה**, כלומר השיטה שעבורו נכתב הקוד – מחלקת הבסיס (בד"כ), אנחנו נגדיר את השיטה ללא ציונים מיוחדים וכך גם השיטה במחלקה הנגזרת (או בציון new). שיטות אלו נקראות סטטיות .

(**קוראים לשיטה לפי טיפוס המשתנה**)

במחלקת האב יוגדר:

public void print1()

{ Console.WriteLine("I am static print1 of myclass1"); }

במחלקת הבן יוגדר:

**new** public void print1()

{ Console.WriteLine("I am static print1 of myclass2"); }

למשל בדוגמא הנ"ל, אם משתנה מסוג מחלקת הבסיס היה מצביע בפועל על מופע של המחלקה הנגזרת, **עדיין** השיטה **המקורית של הבסיס** היתה זאת שנקראת.

**שיטות וירטואליות:**

אם אנחנו רוצים שבמקרה של התנגשות עבור שיטה מסוימת השיטה שתורץ תהיה לפי מה שמשתנה היחס **מצביע עליו בפועל** אנחנו (בד"כ) נגדיר את השיטה של מחלקת הבסיס כ-virtual וזו של המחלקה היורשת כ-override (**קוראים לשיטה לפי המופע שיצרנו בפועל**)

במחלקת האב יוגדר:

**virtual** public void print2()

{ Console.WriteLine("I am virtual print2 of myclass1"); }

במחלקת הבן יוגדר:

**override** public void print2()

{ Console.WriteLine("I am override print2 of myclass2"); }

**הערות:**

1. ניתן לקטוע את שרשרת ההורשה ע"י ציון המילה new במקום override (במחלקה היורשת).

2 . אם בקריאה ממופע של מחלקה נגזרת, נקראה פונקציה של הבסיס, ובאותה הפונקציה יש קריאה ממנה לפונקציה שיש לה override בנגזרת, אז היא זאת שתיקרא ממנה.

**רשומות structs:**

**1**. זהו משתנה ערך ולא יחס, רשומה מוקצאת על המחסנית ולא על ה-heap, בגלל זה ההקצאה והשחרור יותר מהירים.

**2**. לא ניתן להגדיר בנאי ללא פרמטרים עבורה.

**3**. כאשר מכריזים על רשומה השטח מוקצה ישירות, ולכן אין צורך לבצע new. אבל ניתן להשתמש בו, מה שהוא יעשה זה **השמה** ולא הקצאת מקום חדש לרשומה, הרשומה תישאר כמו שהיא היתה רק הערכים שלה ישתנו.

**4**. העברת רשומה לפונקציה, תמיד נעשית **by-value**.

**5**. העבודה עם כל המשתנים והמתודות הם בדיוק כמו במחלקה (מגדירים עבורה בנאי כמו שם הרשומה וכו').

**6**. מבנה לא יכול לרשת ממבנה אחר או מחלקה, וכמו כן מחלקה לא יכולה לרשת ממבנה. לעומת זאת הוא מבנה יכול **לרשת מממשק**.

**הגדרה**:

public structשם רשומה

{ … }

**ממשקים Interfaces:**

**1**. חברי ממשקים יכולים להיות רק הצהרות ללא מימוש של: מתודות, get ו-set , סדרנים.

**2**. חברי ממשק (כלומר המתודות הלא ממומשות שלה וכו' ) חייבים להיות כתובים כ-public למרות שהמילה הזו לא מופיעה בהגדרת הממשק.

**3**. מחלקה יכולה "לרשת" יותר מממשק אחד, והיא מתחייבת לממש אותו (בהורשה).

**4**. ניתן לממש פולימורפיזם ע"י ממשק, למשל ההגדרה הבאה:

I[] c = new I[4];

כאשר I הוא טיפוס ממשק, אך ההשמות עצמן לתוך התאים במערך תיעשה ע"י השמה של מופע (ע"י new) של מחלקה שמממשת את אותו הממשק (חשוב: לא ניתן לעשות new למשתנים מסוג הממשק). כמו כן בצורה דומה ניתן להגדיר משתנה מסוג הממשק ואז לבצע לתוכו השמה של מופע שממש את אותו ממשק.

**הגדרת הממשק:**

interface שם ממשק

{

…

}

**ההגדרה עבור המחלקה המממשת:**

class שם מחלקה : שם ממשק1, שם ממשק2

{

..

}

**הערה**: ניתן לרשת מחלקה אחת וכמה ממשקים, תמיד המחלקה תהיה ראשונה וכל השאר ממשקים.

הבעיה היא שלא ניתן לדעת אם אנחנו יורשים, למשל: מ-3 ממשקים או מ-2 ממשקים ומחלקה.

**תכנית ביותר מקובץ אחד:**

מחלקה חייבת להיות מוגדרת כ-public כדי להיות מוכרת מחוץ לקובץ שלה.

**מחלקות מופשטות abstract classes:**

**1**.מחלקה אבסטרקטית מוגדרת ע"י מילת המפתח abstract יחד עם ההכרזה על שם המחלקה, דבר המקנה לה גם את הזכות להכריז על מתודות שלה כעל פונקציות אבסטרקטיות, גם כן בעזרת מילת המפתח abstract.

**2**.במובן אחד לפחות מחלקות אבסטרקטיות הן מחלקות לכל דבר: אם מחלקה יורשת ממחלקה אבסטרקטית היא לא יכולה לרשת ממחלקה נוספת.

**3**. לא ניתן ליצור מופע מסוג המחלקה האבסטרקטית (ניתן להגדיר משתנה מסוג אותה מחלקה, אבל לא לעשות לבצע עליו new).

**4**.מבחינת הקוד שכתוב בה דומה למחלקה רגילה + תוספת של הצהרות על פונקציות שלא מומשו.

**5**. פונקציה חא מממושת בה יכולה להיות בשימוש במחלקה הזו.

**6**. **כלל 4 שכתבתי לעיל עבור ממשקים** תופס באופן דומה עבור מחלקה אבסטרקטית

**הגדרה**:

**abstract** class שם המחלקה

{

string name;

public **abstract** double Area(); //מתודה ללא מימוש

public **abstract** void Show();// כנ"ל

public void print()

{

Console.Write(name+":");

Show();

Console.WriteLine(", Area = {0}", Area());

} // print

}

ואצל המחלקה היורשת שמממשת את הפונקציות האבסטרקטיות, ההגדרה תיראה כך:

…

public **override** double Area(){…}

public **override** void Show(){..}

**חריגות:**

**1**.חריגה תיוצג ע"י אוביקט מטיפוס Exception או אוביקט נגזר ממנו.

**2**.אם קוד רוצה ליזום חריגה בעקבות שגיאה שנתקל בה, הוא עושה זאת ע"י מילת המפתח throw.

**3.**אם קוד רוצה להריץ קוד העשוי לגרום לחריגה ולשלוט בתגובה ממנה, עליו להשתמש במכניזם של try ו-catch, יש היררכייה של catch מהספציפי ביותר ועד הכללי ביותר שיהיה בסוף הבלוק.

**4**.הקוד שיגיב לחריגה יהיה העמוק ביותר שערוך לכך. הכוונה היא שבלוקים של try-catch יכולים להיות מקוננים.

**5**.ניתן שיהיה catch בלי כלום – הוא יתפוס כל דבר.

**6**. אם יש סוג חריגה שלא הצלחנו לתפוס (למשל בגלל לא הגדרנו), אז התכנית תעוף.

7. ניתן גם לזרוק מחדש את החריגה בבלוק ה-catch שמטפל בה כרגע, ואז הטיפול בחריגה יתבצע בכמה רמות.

**דוגמאות**:

1. יתפוס כל דבר:

try

{ ….. }

**catch**

{ System.Console.WriteLine("Exception Handled!"); }

**finally** {…}

1. ניסיון תפיסת מס' סוגים של חריגות:

**try** { …}

**catch**(NullReferenceException e1)

{ System.Console.WriteLine("Exception A Message: {0}", e1.Message); }

**catch**(Exception e2)

{ System.Console.WriteLine("Exception A Message: {0}", e2.Message); }

1. הגדרת מחלקה חדשה של חריגה:

public class *InvalidAgeException***: Exception**

1. זריקת (יצירת) חריגה:

if (( value < 0) || (value > 200))

**throw** new *InvalidAgeException*(value);

כאשר value זה הערך שנתנו לבנאי של המחלקה של *InvalidAgeException* (מחלקה נגזרת של חריגה).

**בלוק finally**:

הוא יתבצע בכל מקרה, גם אם לא התרחשה חריגה (וגם אם כן, אז מיד אחרי שעברנו את ה-catch המתאים). הערה: אם התבצע return לפני ה-finally, עדיין הוא יתבצע הבלוק הנ"ל.

**אפליקציות מרובות קוד:**

System.Diagnostics.Process.Start("hello2a.exe");

הרצת התוכנית הזו תגרום לפתיחת חלון חדש שבו תרוץ התוכנית הקודמת בעוד שהתוכנית הזו תרוץ באופן בלתי תלוי (כמו שני תהליכים שונים). יהיה אפשר לעבוד עם האפליקציה המקורית והחדשה לסירוגין או לעבוד עד הסוף עם אחת מהן, לא משנה איזה.

AppDomain ad = AppDomain.CreateDomain("my new domain");

ad.ExecuteAssembly("hello3.exe");

שימוש ב-AppDomain תריץ קובץ ביצועי (.exe) אחר במסגרת **אותו תהליך** מבחינת מערכת ההפעלה. מבחינה המשתמש זה יבוא לידי ביטוי בכך שהקובץ הנפרד ירוץ באותו חלון, תוך השהיית התוכנית הראשית.

הערה: ההתנהגות של א' למשל עבור חריגות ב"ת בין החריגות, ואילו של ב', חריגה באחת מהתוכניות תביא להפסקת שתי האפליקציות (כי רצות על אותו התהליך).

**חוטים Thread:**

**הגדרה והרצת חוט:**

*System.Threading;*

*...*

*ThreadStart myThreadStart = new ThreadStart(פונקציה);//*

ThreadStart: זהו delegate (פוינטר לפונקציה) שתפקידו להצביע על המתודה שהיא נקודת ההתחלה של ה-Thread.

*Thread myThread = new Thread(myThreadStart);*

Threadזאת מחלקה המאפשרת לנו להגדיר משתני מופע מסוג משימות. הבנאי של Thread יקבל כפרמטר ThreadStart שיקבע את הקוד שהוא נקודת ההתחלה של ה-Thread.

*myThread.Start();*

קריאה ל-Start תחזור מיד או כמעט מיד לקוד שקרה לה, לא בהכרח לאחר שה-Thread סיים את פעולתו.

הערה: ניתן גם לוותר יצירת הדלגט, ולתת ל-Thread כפרמטר את שם הפונקציה שרוצים להריץ.

הערה: אם חוט אחד עף בגלל חריגה, זה לא ישפיע על חוט אחר מאותה קבוצה שלו.

הערה – אם נרצה לחלק עבודה של פונקציה לשני חוטים וכל אחד עובר על חלק אחר ואחרי צריך לחבר בין התוצאות. ניצור מופע לכל חוט ולכל אחד יהיה משתנה res משלו, כאשר הפונקציה שהחוט ייקבל כפרמטר תהיה זהה עבור שניהם וישמור את התוצאה ב-res עבור המופע שלו. ובסיום ב-main נקרא ל- o.res של כל חוט ונחבר בינהם.

**ניתן גם לממש חוט ע"י כך שמחלקה מסויימת תירש ממחלקת Thread.**

**1**.פעולות סנכרון חוטים

**א**. סנכרון חוטים (דומה לsync- ב-java) :

lock(משתנה מחלקה שרוצים לנעול) {…}

והפעולות:

wait(object)//

pulse(object)//אקוולנטי לסיגנל

PulseAll(object);//כנ"ל

קודם משתמשים ב-lock ובפנים לפי תנאי עושים wait פקודה ואח"כ pulse.

**ב.** סנכרון ע"י שימוש במחלקה: ReaderWriterLock **(דוגמא מלאה בעמ' 165)**

המחלקה ReaderWriterLock מאפשרת לנעול משתנה לכתיבה אבל להותיר את המשאב פתוח לקריאה בו זמנית. זאת משום שאין סיבה לאסור גישות בו זמניות למשתנה אם כל הגישות הן לקריאה בלבד. אולם אם יש בקשה לכתיבה למשאב, היא תמתין לסיום הקריאות הקיימות, לא תתיר קריאות נוספות, ולבסוף תיתן בלעדיות ל-Thread הכותב.

נוהל העבודה היא יצירת משתנה ReadWriterLock, ביצוע AcquireReaderLock או AcquireWriterLock עליו תוך פירוט זמן המתנה במילי שניות (אפשר אינסוף), וביצוע ReleaseReaderLock או RleaseWriterLock

     int initial = 0;  
        ReaderWriterLock rwl = new ReaderWriterLock();  
      public void myRead(object threadName)  
        {  rwl.**AcquireReaderLock**(Timeout.Infinite);//  
            …//פעולת קריאה  
            rwl.**ReleaseReaderLock**(); }  
        public void myWrite()  
        { rwl.**AcquireWriterLock**(Timeout.Infinite);   
            initial++; //פעולת כתיבה

rwl.**ReleaseWriterLock**(); }

**ג.** סנכרון ע"י שימוש במחלקה System.Monitor:

**Monitor.Enter(object); //התחלה**

… //כל הדבר הנ"ל אקוולנטי לבלוק של **נעילה**, רק בלי סוגריים מסולסלות

**Monitor.Exit(object);//סוף הנעילה**

כאשר קריאה ל-wait() תשחרר את הנעילה עד שיבוצע pulse(), וקיימת גם PulseAll().

public bool Dequeue( out T t )

..

**lock** ( \_Key ){

**while** ( !\_Quit && \_Queue.Count == 0 ) M Monitor.**Wait**( \_Key );

...

t = \_Queue.Dequeue();

Monitor.**PulseAll**( \_Key );

public bool Enqueue( T t )

{

**lock** ( \_Key )

{

**while** (\_Queue.Count >= \_Size )

Monitor.**Wait**( \_Key );

...

\_Queue.Enqueue( t );

Monitor.**PulseAll**( \_Key );

**SyncRoot:** זה משתנה מחלקה המיוחד שנמצא במחלקות כמו תור וכו' (כמובן שתמיד ניתן להגדיר במחלקה חדשה משהו דומה) ועליו נהוג לבצע את ה-lock במקום על this שנעילה שלו עלולה להביא ל-deadlock.

**2. פעולות נוספות על חוטים:**

Thread.sleep(1000);//זה שווה לשניה אחת בה החוט ילך לישון

Thread a\_thread = new Thread(astart\_point);

a\_thread.Join();//המתנה לחוט הנ"ל

a\_thread.Abort();//חיסול של החוט הנ"ל

a\_thread.Priority = Thread.Priority.Highest;//נתינת עדיפות עליונה לחוט(שימושי לפני הריצה)

אלה העדיפויות: Lowest, BelowNormal, Normal, AboveNormal, Highest.

**הערות**:

שחוט מפעיל פעולות על עצמו, זה יעשה ע"י קריאות למתודות סטטיות, וכאשר חוט מבצע פעולות על חוט אחר זה יעשה ע"י קריאה למתודה על המופע של החוט שעליו מבקשים לבצע את הפעולה.

שיתוף מידע בין חוטים, 2 אפשרויות עיקריות:

א. להגדיר משתנים סטטיים במחלקה כך שיהיו משותפים בין החוטים.

ב.להקצות מופע של מחלקה שתהיה משותפת בין החוטים ולהעביר את כתובת המופע של לחוטים.

**התמרות casting:**

**א**. **casting על פרימיטיביים**:

השמות של אי-אובדן מידע על פרימיטיביים, כמו למשל התמרה מ-float ל-double, מ-int ל-long וכו'.

ואם רוצים לעשות cast "מסוכן", רושמים במפורש(עם סוגריים):

double d=5.5;

f = (float) d; *//cast to 5*

**ב**. **casting על משתני ייחוס**:

מתאפשר רק עבור מחלקות עם אב משותף.

1. **Upcasting**: המרה ממחלקת בן למחלקת אב שלה (כלשהי), אם יש שיטות שנדרסות(override) ע"י הבן, הריצה של השיטה תהיה לפי הבן וזה יהיה תמיד לפי המופע עצמו (זאת תהיה התמרה משתמעת כלומר עם סימן **=**) ההמרה ניתנת רק אם כל השיטות שבבן קיימות אצל האב.

2. **Downcasting**: כאשר משתנה מסוג האב, מצביע בפועל למופע של הבן, טוב כאשר רוצים לבצע שיטות שיש רק בבן , ההתמרה תתבצע עם סוגריים (מפורשת).

דוגמא:

כאשר A מחלקת הבסיס ו-C יורש מ-A.

A a = new C();// Upcasting

C c = (C)a; // Downcasting

אם למשל היינו עושים (C)a כאשר בפועל a לא היה מחזיק מופע של C, היינו מקבלים שגיאת זמן ריצה.

B b = new B();

A a = b;

~~C c; c = a; // Generates compiler error~~

~~c = (C) a; // Generates run time error~~

**הפקודה is:**

בודקת האם האובייקט a מכיל מופע של מחלקה A:

if(a is A);

**תכנות גנרי:**

**א.** תכנות גנרי ע"י casting, ללא הגדרה של מחלקה מסוג T:

מחלקת object יכולה להצביע גם על פרימיטיביים

int i = 42;

object o = i; //box

int j = (int)o; //unbox

**דוגמא עם מערך:**

נגדיר מערך מסוג Object (שיוכל להכיל כל סוג אובייקט ,כולל פרימיטיביים)

כאשר הפונקציה שמבצעת השמה למערך הנ"ל, היא תקבל פרמטר מסוג object

כאשר נוציא איבר מהמערך, ונרצה למשל להדפיסו(או להשתמש בשיטות שלו), נבדוק ע"י אופרטור is את סוג המופע לפני כן, ואחרי שקיבלנו את סוגו נבצע לו casting.

**ב.** תכנות גנרי ע"י casting מחלקה מסוג T:יכול להחזיק רק סוג משתנה כפי שהוגדר עבורו

נגדיר את המחלקה מסוג T

נגדיר מערך מסוג T

תכנית שימוש ספציפית כמו Double תהיה מחלקה נפרדת והיא תהיה הטיפוס T ברגע הריצה.

נגדיר בתכנית הראשית (main), מערך מסוג ספציפי(כולל פרימיטי)

הפרמטר לפונקציה שמבצעת השמה למערך, תהיה עם משתנה מסוג T

בשליפת איבר מהמערך, אין צורך כלל ב-cast, נחזיר ערך מסוג T.

הערה: אם בחרנו לממש עם מערך מסוג object במקום T, נהיה חייבים שהפונקציה שמבצעת השמה למערך תקבל פרמטר מסוג T, ובשליפת איבר מהמערך, נבצע casting לסוג T.

static void Swap <T>(ref T lhs, ref T rhs)

{

T temp;

temp = lhs;

lhs = rhs;

rhs = temp;

}

int a = 1,b = 2 ;

**Swap<int>(ref a, ref b);**

**public class GenericList <T>**

{

void Add(T input) { }

}

**static void Main()**

{

GenericList<int> list1 = new GenericList<int>();

GenericList<ExampleClass> list3 = new GenericList <ExampleClass>();

}

תוספות:

1. מספר רנדומלי:

Private int RandomNumber(int min, int max)  
{  
Random random = new Random();  
return random.Next(min, max);   
}

Random rnd = new Random();

int month = rnd.Next(1, 13); // creates a number between 1 and 12

int card = rnd.Next(52); // creates a number between 0 and 51

2. enum:

enum Status {Ready, NotReady};//מוגדר כמו משתנה מחלקה

..

Status myStatus = Status.NotReady;

3. תור מוכן:

Queue queue;

Queue queue = new Queue();

queue.Enqueue(object);

queue.Count //מס איברים בתור

4. טבלת האש:

Hashtable bank = new Hashtable();

bank.Add(Shimon, 400);

מה נשאר להוסיף:

צריך להוסיף: על מערכים ומופעים עם ניו

ועל העמסת טופרטורים.

Dll-

ברגע שעושים שימוש ב- namespace ויש שם מחלקות. אז תוכנית אחרת יכולה לעשות שימוש בקובץ הזה (שקומפל ל-dll) ע"י שימוש ב-using name כאשר name הוא שם ה-namespace של קובץ ה-dll. ואז ניתן להשתמש במחלקות או פונקציות זטטיות משם.

רפלקציה – מאפשר לעשות שימוש ב-dll מבלי לבצע קימפול מחדש של כל התכנית, בעצם זה plugin.

Reflection - בשפות תכנות מונחות-עצמים כמו Java, שימוש ב-reflection מאפשר בדיקה בזמן ריצה של מחלקות, ממשקים, שדות ומתודות, מבלי לדעת את השמות שלהם בזמן הידור.