

חריגות (Exceptions)

קיצור מונחים:

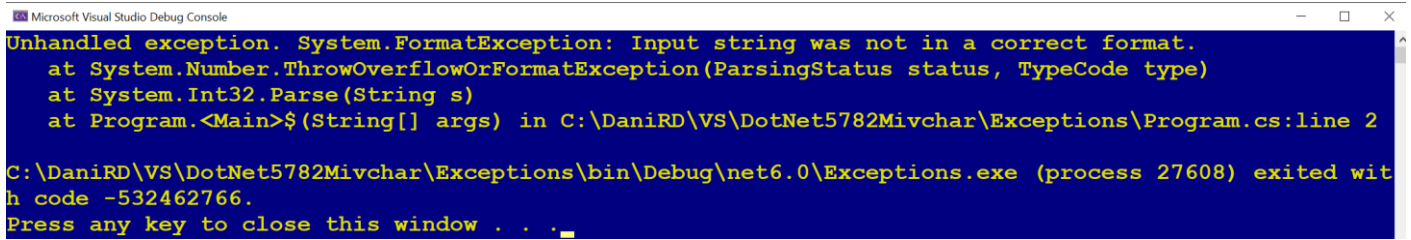
חריגה – תוצאה של פסיקת חומרה (**Interrupt**) או פסיקה פרוגרמטית המצביעה על תקלה בחומרה או בתוכנה. דוגמאות לפסיקת חומרה – תקלת פלטקלט, חלוקה באפס. בהמשך נקרא לפסיקת חומרה פשוט "פסיקה" ולפסיקת תוכנה – "חריגה", אבל לפעמים כשנגיד "חריגה" נתכוון לשניהם – הכול לפי הדיון הספציפי.

אירוע של חריגה

אם במהלך ריצת התוכנית קורת פסיקה או חריגה, ריצת התוכנית מופסקת ומערכת הפעלה (או .NET). מעביר את הבקרה לתוכנית טיפול בחריגה. בדרך כלל מערכת ההפעלה תוציא פלט של חריגה על פי צורת הריצה של התוכנית – הדפסה לקונסול ואו לתוך קובץ לוגים, הקפצת חלון המתריע על התקלה עם מידע עליה וכדומה. התוכנית המקורית לא תמשיך להתבצע. בדרך כלל אפשר להבין מהמידע על החריגה מה סוג התקלה עם פירוט נוסף, איפה היא קרתה (כולל שרשרת הקריאות לפונקציות של התוכנית, כולל מספרי שורה). למשל נריץ את הקוד הבא (בתצורת תוכנית קונסול):

```
int num = int.Parse("aaa");
```

דבר ראשון, נקבל בקונסול פלט דומה למה שרואים בצילום המסך הבא:

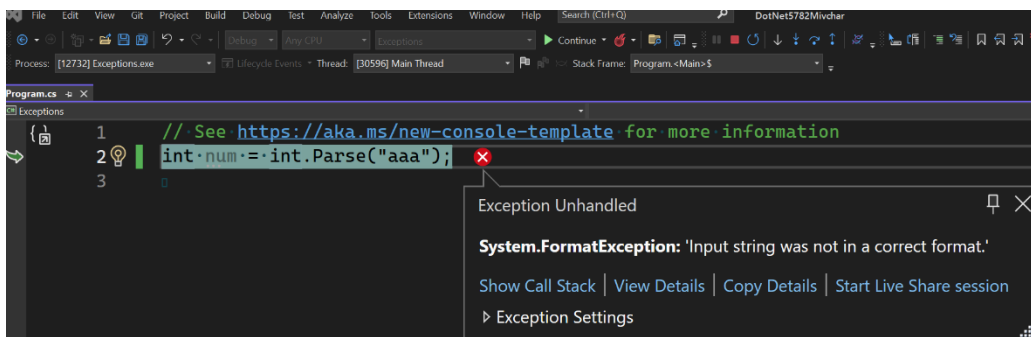


```
Microsoft Visual Studio Debug Console
Unhandled exception. System.FormatException: Input string was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
   at System.Int32.Parse(String s)
   at Program.<Main>$(String[] args) in C:\DaniRD\VS\DotNet5782Mivchar\Exceptions\Program.cs:line 2
C:\DaniRD\VS\DotNet5782Mivchar\Exceptions\bin\Debug\net6.0\Exceptions.exe (process 27608) exited with code -532462766.
Press any key to close this window . . .
```

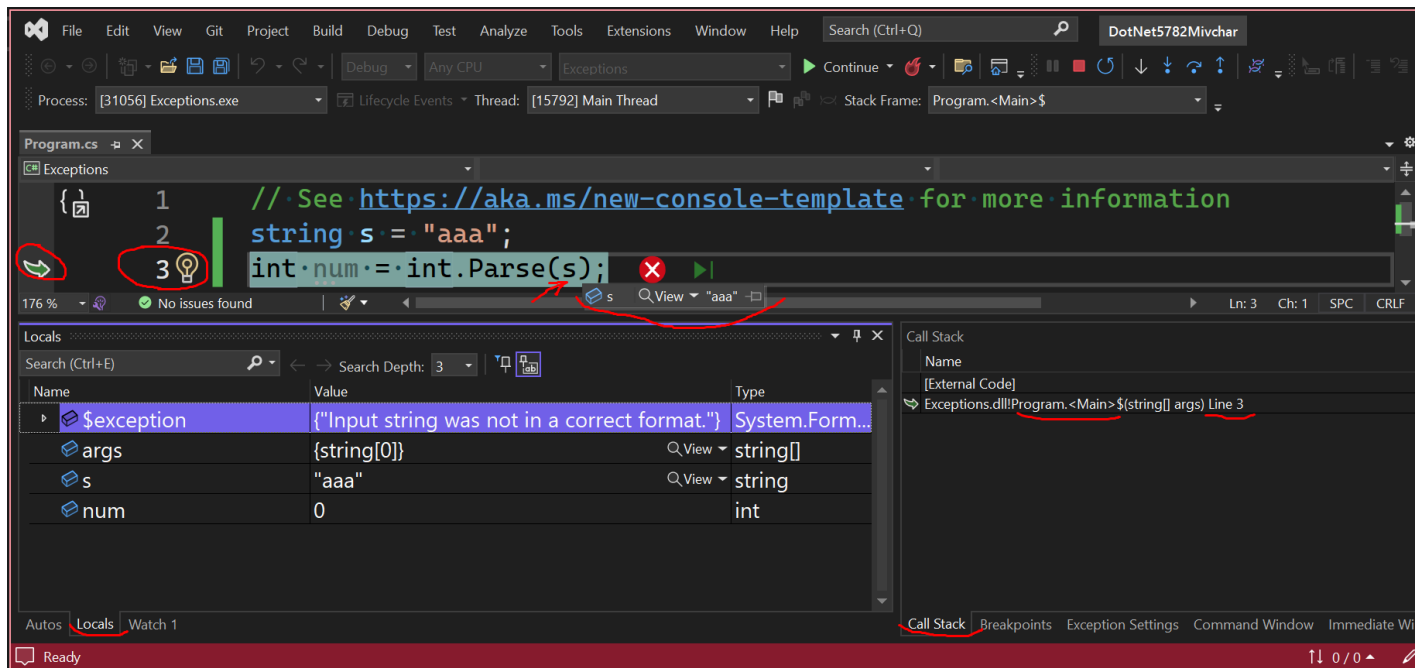
ממה שהוצג נדע שהתקלה היא בעיית פורמט (**System.FormatException** מסוג חריגה) ובנוסף קיבלנו פירוט שהבעיה נובעת מכך שמחרוזת שקיבלנו איפשרה (תכף נראה איפה) לא הכילה מידע בפורמט הנכון. לאחר מכן נראה שורות של מחסנית קריאות הפונקציות, חל מהפונקציה הראשית של התוכנית (הכי למטה) עד הפונקציה שיצרה את החריגה או גרמה לפסיקה. בכל שורה מוצג השם המלא של הפונקציה, כולל שם של **namespace**, שם המחלקה ושם הפונקציה עצמה – מופרדות ע"י נקודה – בפורמט המוכר, חתימת הפונקציה מבחינת הפרמטרים (שמותם וסוגיהם) ומספר שורה (שלפעמים לא יוצג בפונקציות של ספריות מערכתיות). במקרה הזה אנחנו רואים שהתקלה קרתה כאשר בתוכנית שלנו (בשורה 12) פנינו לפונקציה **Int32.Parse**. נתבונן בשורה הזאת ונבין שנסינו להפוך מחרוזת טקסט שאיננה כוללת רק ספרות למספר – לעל זה מעידה החריגה. **נ.ב.** אתם זוכרים שהגדרת **int** הינה בפועל **Int32**.

אבל מה קורה עם הדבר לא ברור מאלינו? למשל, היינו מעבירים ל-**Parse** הזה משתנה מחרוזת שקיבלנו ממקום אחר – אולי מקלט. איך נדע מה היה בתוך המחרוזת?

אם נבצע הרצה עם **Debug**, ניכנס בעקבות זריקת הרחגיה למצב דיבאגינג (תהליך איתור וניפוי שגיאות תוכנה) בויזואל סטודיו כאשר התוכנית עצרה בשורה שקרתה בה תקלה. נוכל לראות אפשרויות לאיתור תקלה וגם לראות את פרטי החריגה – אם נלחץ על הקישור "**View Details**":



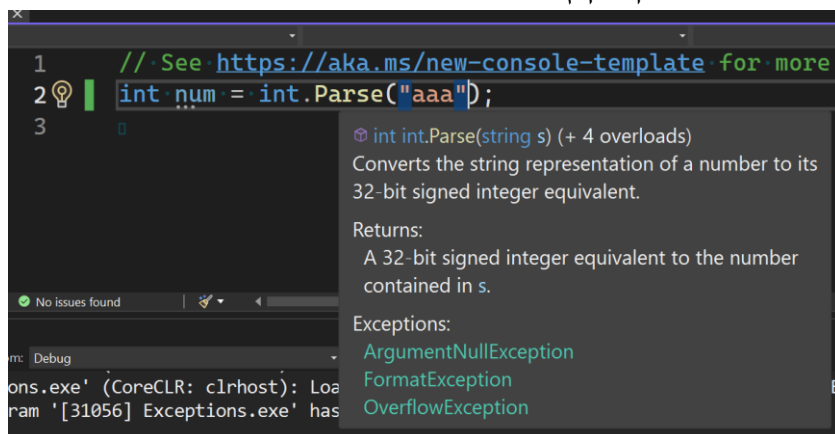
נוכל גם לעין בערכי המשתנים ולבדוק דברים נוספים כרצונכם. למשל – נעצור את סמן העכבר מעל המשתנה בקריאה ל- **Parse** ונבחין בערך שלו בפועל (נראה אותו גם בחלונית של משתנים "**Locals**" למטה) ונוכל לראות בחלונית של מחסנית זימונים (**Call Stack**) את סדר הזימונים של המתודות והשורה שבה קרתה התקלה:



תפיסת חריגה

ברוב המקרים לא נרצה שהתוכנית שלנו תקרוס באמצע, ונרצה להתייחס לחריגות שעלולות לקרות בתוך הקוד שלנו ולתת טיפול הולם בלי להפסיק את ריצת התוכנית. בשביל המשימה הזאת קיים בשפת C# מנגנון תפיסת חריגות.

בואו נעצור את סמן העבר מעל פונקציית **Parse** ונראה מכתבו בחלון שקפץ:



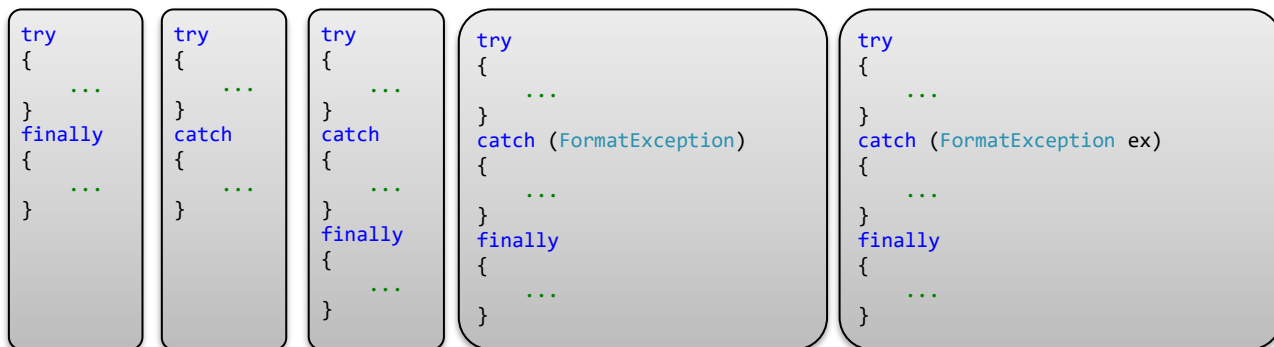
```
1 // See https://aka.ms/new-console-template for more
2 int num = int.Parse("aaa");
3
```

int int.Parse(string s) (+ 4 overloads)
Converts the string representation of a number to its 32-bit signed integer equivalent.

Returns:
A 32-bit signed integer equivalent to the number contained in s.

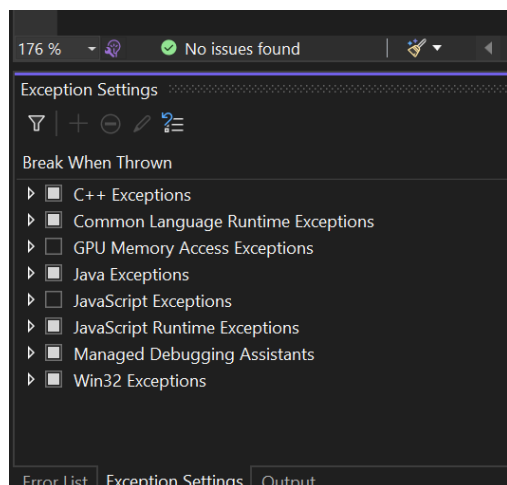
Exceptions:
ArgumentNullException
FormatException
OverflowException

נראה בחלון הזה רשימת החריגות (**Exceptions:**) שעלולות להיווצר בתוך הפונקציה. זה כבר אומר לנו שיש סיכוי שהתוכנית שלנו עלולה לקרוס בקריאה לפונקציה הזאת. מה נעשה? נשתמש בתכונה של שפת C# הנקראת בלוק **try**. בלוק **try** מתחיל במילת מפתח **try** וכולל אחריו בלוק בסוגריים המסולסלות { } שיכיל בתוכו את הקוד העלול לייצר או לגרום לחריגה. בלוק **try** חייב לכלול אחריו אחד או יותר בלוקים של **catch** ואו בלוק **finally** – שני המילים הינן גם מילות מפתח ב-C#. בלוק **catch** יכול לכלול גם פרמטר לאחר מילת המפתח. הפרמטר יהווה בעצם את שם מחלקת החריגה שאנחנו רוצים לתפוס, וגם יכול לכלול שם משתנה שבו נקבל את פרטי החריגה ונוכל להשתמש בהם.

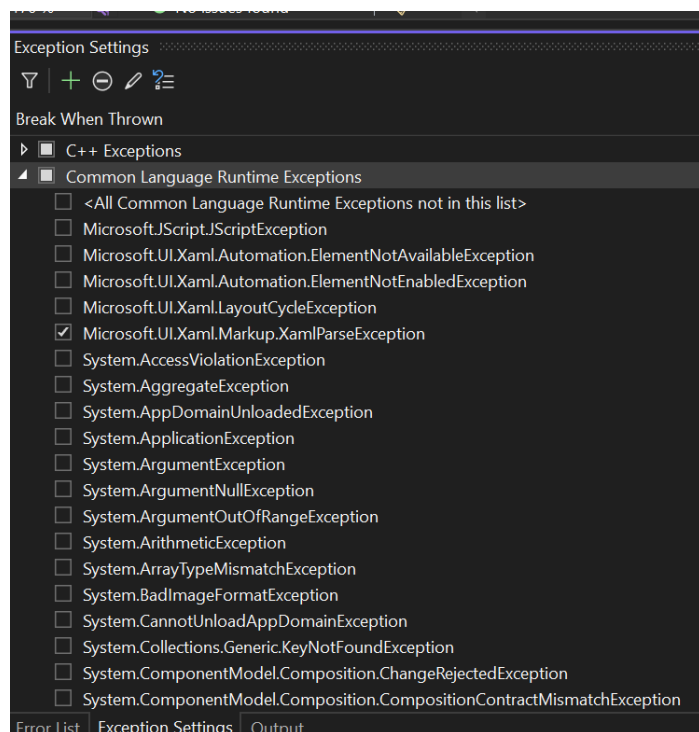


בלוק **try** בעצם יכול בתוכו את הקוד שמתוכו יכולה לקרות החריגה – והכלל הוא שאנחנו נשתדל לכלול בתוכו את הקוד המינימלי האפשרי – וזאת בשביל לצמצם אי הודאות של מקור החריגה ככל הניתן. בדרך כלל לא נכלול יותר משורת קוד בודדת בתוך הבלוק הזה.

כתבתי קודם: "אחד או יותר בלוקים של **catch**". מה הכוונה? עכשיו נכיר מה זה בעצם מחלקות החריגות. בספרית המערכת של C# כלולה המחלקה הבסיסית של כל החריגות – מחלקת **Exception**, או בשמה המלא – **System.Exception**. כל סוג חריגה מוגדר על ידי מחלקת חריגה נפרדת. כל מחלקות החריגה יורשות באופן ישיר או עקיף מהמחלקה הזאת. כל מחלקות החריגות מסתיימות במילה **Exception**. להבדיל מסביבות אחרות (למשל – Java) במיקרוסופט החליטו לשטח את עץ המחלקות של חריגות, ובדרך כלל אם החריגה לא יורשת מהמחלקה הבסיסית **Exception** באופן ישיר אז מחלקת האב שלה יירש ממנה ישירות. ניתן לראות את כל החריגות המוגדרות ב-.NET בתוך כלי **Exceptions** – אפשר לפתוח אותו ע"י לחיצה **Ctl+Alt+E** בויזואל סטודיו:



החריגות של .NET. נמצאות תחת **Common Language Runtime Exceptions** (זוכרים איך נקראת המכונה הוירטואלית של .NET?). נלחץ על סימן של משולש לידה. רוב החריגות הנחוצות לנו בשלב הזה נמצאות בספרית המערכת **System** – נעניין בחלק מהרשימה שמתחיל מ-**System**:



אם נשתמש בפרמטר של בלוק **catch** – בעצם נגדיר מסנן – רק החריגה מהסוג שנקטנו בפרמטר או חריגות שיורשות ממנה תגרומנה לביצוע אותו הבלוק. אם נוסיף מספר בלוקי **catch** – הם ייבדקו אחד אחרי השני והבלוק הראשון שעונה על החריגה שקרתה יתבצע – ורק הוא. לכן נרשום בלוקים של חריגת אב וחריגות שיורשות ממנה – בלוק חריגת האב יהיה אחרי הבלוקים של חריגות היורשות. ולכן הבלוק שיכלול את החריגה הבסיסית **Exception** יהיה הבלוק האחרון בשרשרת.

נ.ב. בלוק ללא פרמטר תופס את כל החריגות שלא יופיעו בשרשרת הבלוקים **catch**. בעצם **catch** ללא פרמטר הינו שווה לבלוק **(Exception) catch**.

נ.ב. ויזואל סטודיו ומהדר C# לא יאפשרו לרשום מסנני **catch** בסדר לא נכון וגם לא יתנו לחזור על אותה חריגה פעמיים – ובעצם לא יתנו לכם לרשום את בלוקי **catch** בסדר לא נכון בטעות.

ואחרון חביב – בלוק **finally** – מתבצע לפני סיום בלוק **try** ו-**catch** בכל מקרה – מיד לפני סיומו. גם במקרה והשתמשם ב-**break, continue, goto, throw** או בלוק **finally** יתבצע מיד לפנייהם. תפקידו העיקרי – לשחרר משאבים שנתפסו לפני כן. למשל – לסגור קבצים שנפתחו לפני כן ואין טעם להחזיק אותם פתוחים בגלל התקלה שקרתה, או הולכים

לסיים את התוכנית בכל מקרה. יחד עם זאת יש פח אחד שלא כדאי לפול לתוכו... עם החריגה שקרתה לא נתפסה באף אחד מבלוקי **catch** – המצב נקרה "unhandled exception" – ביצוע הבלוק **finally** תלוי בקונפיגורציה ומימוש של המערכת ו- CLR... יותר פרטים (מי שירצה) תמצאו במאמר באתר מיקרוסופט למפתחים בקישור הבא (המאמר באנגלית):
<https://learn.microsoft.com/en-us/archive/msdn-magazine/2008/september/clr-inside-out-unhandled-exception-processing-in-the-clr>
 לאחר כל הבלוק התוכנית ממשיכה כרגיל – אלא השתמשתם בתוך בלוק **catch** בפעולה שמסיימת את הפונקציה – **return** או **throw** (נלמד את זה עוד מעט). דוגמה קצרה למה שלמדנו עד כה:

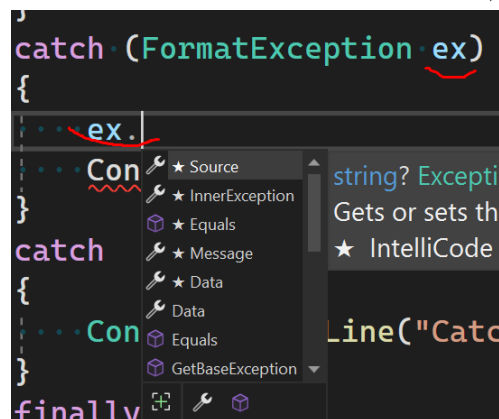
```
try
{
    int num = int.Parse("aaa");
}
catch (ArgumentNullException)
{
    Console.WriteLine("Catch ArgumentNullException");
}
catch (FormatException)
{
    Console.WriteLine("Catch FormatException");
}
catch
{
    Console.WriteLine("Catch");
}
finally
{
    Console.WriteLine("Finally");
}
Console.WriteLine("Did it!");
```

Microsoft Visual Studio Debug Console

```
Catch FormatException
Finally
Did it!
```

והפלט כדלקמן:

בחזרה לבלוק **catch**. מה אפשר לעשות בתוכו? בעצם כל מה שאתם רוצים. אפשר להדפיס את המידע על החריגה ולסיים את התוכנית. אפשר גם לנסות להפנות את התוכנית לדרך שונה שתוכל לבצע את המשימה תוך עקיפת הבעיה. אפשר גם לבקש מהמשתמש קלט מחדש עם הסבר על הבעיה שנגרמה לפני כן. אבל איך נקבל פרטים על החריגה בנוסף לסוג שלה? בשביל זה נוכל להוסיף לפרמטר של **catch** שם משתנה ולהשתמש בו בתוך הבלוק:



בנוסף לפונקציות הרגילות שנורשו מ-**Object** נגלה מספר שדות. בואו נתבונן בהגדרת מחלקת **Exception** ונרחיב על חלק מהשדות:

```

... public class Exception : ISerializable
{
...     public Exception();
...     public Exception(string? message);
...     public Exception(string? message, Exception? innerException);
...     protected Exception(SerializationInfo info, StreamingContext context);

...     public virtual string? StackTrace { get; }
...     public virtual string? Source { get; set; }
...     public virtual string Message { get; }
...     public Exception? InnerException { get; }
...     public int HRESULT { get; set; }
...     public virtual IDictionary Data { get; }
...     public MethodBase? TargetSite { get; }
...     public virtual string? HelpLink { get; set; }

...     protected event EventHandler<SafeSerializationEventArgs>? SerializeObjectState;

...     public virtual Exception GetBaseException();
...     public virtual void GetObjectData(SerializationInfo info, StreamingContext context);
...     public Type GetType();
...     public override string ToString();
}

```

קודם כל – שימו לב על ארבעת הבונים של המחלקה ובשדה **InnerException** – ניתקל בהם עוד.

- שדה **Message** מכיל הודעה נוספת שמפרטת את הסיבה לחריגה שקרתה.
 - שדה **Source** מכיל שם האפליקציה או שם העצם (אובייקט) שגרם לחריגה.
 - שדה **StackTrace** מכיל את שורות של מחסנית קריאות הפונקציות כמו שכבר ראינו קודם
 - שדה **TargetSite** מכיל את המידע (כבר ראינו ב-Reflection) על המתודה (הפונקציה) שזרקה את החריגה (נלמד בהמשך)
 - שדה **Data** מכיל אוסף זוגות של מפתח/ערך המספק מידע נוסף על החריגה
- שימו לב שהמחלקה הבסיסית כוללת יישום משלה לפונקציה **ToString** שמדפיסה את המידע בפורמט שכבר ראינו קודם. מידע מפורט וכמה דוגמאות לשימוש: – באתר מיקרוסופט למפתחים:

<https://learn.microsoft.com/en-us/dotnet/api/system.exception?view=net-7.0>

<https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/exception-handling>

זריקת חריגה

פסיקה פרוגרמטית הינה תוצאת זריקה של חריגה יזומה מתוך תוכנה. יזימת (=זריקת) חריגה ב-C# מתבצעת ע"י אופרטור **throw** (מילה שמורה). לאופרטור הזה יש פרמטר אחד – אובייקט החריגה. לכן בד"כ נראה משתנה מסוג חריגה (נדבר על זה בהמשך) או יצירת מופע חדש מתוך אחת המחלקות של חריגות בו במקום.

דוגמה לשימוש במשתנה עם חריגה : דוגמה ליצירת מופע חריגה בו במקום :

```
throw new SomeException();
```

```
SomeException ex;  
...  
throw ex;
```

זריקת חריגה יכולה להתבצע גם מתוך בלוק **catch** של תפיסת חריגה. ובמקרה כזה יכולה להתבצע ללא פרמטרים. הסיבה לכך היא שבלוק **catch** מגדיר באיזו חריגה הוא מטפל. החריגה הנזרקת במצב זה היא אותה החריגה המטופלת בבלוק **catch** הנוכחי. בדוגמה למטה תיזרק חריגה **SomeException** (שימו לב ששלושת הדוגמאות שוות) :

```
try  
{  
    ...  
}  
catch (SomeException ex)  
{  
    ...  
    throw;  
}
```

```
try  
{  
    ...  
}  
catch (SomeException)  
{  
    ...  
    throw;  
}
```

```
try  
{  
    ...  
}  
catch (SomeException ex)  
{  
    ...  
    throw ex;  
}
```

מתי נרצה להשתמש בזריקת חריגה? למשל, לפעמים אין לנו דרך אחרת (ערך מוחזר או פרמטר) להודיע לפונקציה הקוראת על בעיה כלשהי. לפעמים תקרה חריגה\פסיקה בתוך הפונקציה שלנו ונרצה לייצר חריגה שונה במקומה. בואו נתמקד בדוגמה האחרונה – החלפת חריגה. במצב הזה אנחנו קודם כל כבר בתוך בלוק **catch** תוך טיפול בחריגה כלשהי. מסיבות כאלה או אחרות של צורכי הפיתוח שלנו נרצה לייצר חריגה שונה מזאת שתפסנו. לפעמים נרצה לייצר אותה כמות שהיא בלי מידע נוסף, ולפעמים נרצה להוסיף מידה בצורת מחרוזת נוספת ואז מידע על החריגה המקורית – נשתמש בבונים שונים ליצירת חריגה חדשה :

```
try  
{  
    ...  
}  
catch (SomeException ex)  
{  
    ...  
    throw new OtherException("Some message");  
}
```

```
try  
{  
    ...  
}  
catch (SomeException ex)  
{  
    ...  
    throw new OtherException();  
}
```

```
try  
{  
    ...  
}  
catch (SomeException ex)  
{  
    ...  
    throw new OtherException("Some message", ex);  
}
```

בדוגמה השלישית – החריגה המקורית תישמר בשדה **InnerException** של החריגה החדשה. ניתן גם ליצור את החריגה החדשה במשתנה נפרד, להוסיף מידע בתוכו (למשל בשדה **Data**) או בשדות אחרים שיכולים להופיע בחריגות ספציפיות.

שימו לב – אמנם ניתן טכנית לזרוק חריגת *Exception*. אבל מיקרוסופט מתנגדים לזה באופן נחרץ ושימוש בה הינה פרקטיקה גרועה מאד. והסיבות הן ברורות: לא ניתן להבין מה בדיוק קרה ולא ניתן לבצע סינון חריגות בבלוק *try*.

בדרך כלל נחפש ונזרוק אחת החריגות שכבר קיימות ב-.NET. מהרשימה שראינו בהתחלה (כלי Exceptions). אבל לפעמים לא נמצא שם חריגה מתאימה. אז נרצה משהו מיוחד לאפליקציה שלנו. במקרה כזה נוכל להגדיר מחלקת חריגה מיוחדת משלנו ומותאמת אישית.

יצירת חריגה מותאמת אישית

בשביל יצירת חריגות משלנו (מותאמות אישית) מצטרך להגדיר מחלקות נפרדות לכל חריגה ספציפית. הינה רשימת הכללים ליצירת מחלקת חריגה, כפי שמופיע באתר של מיקרוסופט:

- תימנעו מהיררכיות עמוקות... זאת אומרת – תירשו מהמחלקה הבסיסית **Exception** (וזאת ההמלצה החמה ביותר של מיקרוסופט) או באחת החריגות הכלליות שיושרות ממנה.
- שם המחלקה יסתיים במילה **Exception**. כמו כל החריגות שראיתם בכלי.
- תעשו את המחלקה serializable. המשמעות של המושג בסביבות .NET ו-Java – סוג מחלקה הניתנת להמרה לזרם ביטים או תווים שניתנים לשמירה וואו לשידור בפרוטוקולי תקשורת. לא נתעמק יותר מידי במושג הזה. רק נגיד שלושה דברים:
- המחלקה תממש אינטרפייס **ISerializable** – בעצם אין לכן שום פעולה לנקוט בשביל זה כי המחלקה הבסיסית של חריגות כבר מממשת את האינטרפייס הזה ואתם תירשו ממנה או מאחד מבנותיה.
- המחלקה תכלול אטריבוט **[Serializable]** לפניה.
- אם אתם מוסיפים שדות משלכם – תוודאו שהסוג שלהם גם מוגדר עם האטריבוט הזה (כל הסוגים הבסיסיים והנפוצים כוללים אותו), או תוסיפו אטריבוט **[NonSerialized]** לפני השדה.

```
[Serializable]
public class MyException : Exception
{
    public MyException() : base() {}
    public MyException(string message) : base(message) {}
    public MyException(string message, Exception inner) : base(message, inner) {}
    protected MyException(SerializationInfo info, StreamingContext context) : base(info, context) {}
}
```

- אם אתם מתכננים שמישהו אחר ישתמש ב- וואו יפתח בעזרת הקוד שאתם כותבים – תיישמו לפחות את ארבעת הבונים הבסיסיים של חריגה כדלקמן (כולל האטריבוט הנ"ל):
- הערה נוספת על הנ"ל – אתם יכולים להוסיף קוד משלכם לבונים האלה
- אתם יכולים להוסיף בונים משלכם שעונים על הצרכים שלכם
- אתם יכולים להוסיף שדות משלכם לפי הצרכים שלכם להוסיף מידע לחריגה
- בדרך כלל תרצו לשכתב (override) פונקצית **ToString** כדי לאפשר פלט חריגה מותאם לצרכיכם.

דוגמה לחריגה מותאמת:

```
[Serializable]
public class OverloadCapacityException : Exception
{
    public int capacity { get; private set; }

    public OverloadCapacityException() : base() {}
    public OverloadCapacityException(string message) : base(message) {}
    public OverloadCapacityException(string message, Exception inner) : base(message, inner) {}
    protected OverloadCapacityException(SerializationInfo info, StreamingContext context) : base(info, context) {}
    // special constructor for our custom exception
    public OverloadCapacityException(int capacity, string message) : base(message) =>
        this.capacity = capacity;
    override public string ToString() =>
        "OverloadCapacityException: DAL capacity of " + capacity + " overloaded\n" + Message;
}
```