## **try Statements and Exceptions** **mencoba Pernyataan dan Pengecualian**

In programming, unexpected situations can arise during execution, such as trying to divide by zero, accessing a file that doesn't exist, or running out of memory. These situations are called **exceptions**. C# provides a structured mechanism to deal with these exceptions using try statements, catch blocks, and finally blocks. Dalam pemrograman, situasi tak terduga dapat muncul selama eksekusi, seperti mencoba membagi dengan nol, mengakses file yang tidak ada, atau kehabisan memori. Situasi-situasi ini disebut pengecualian. C# menyediakan mekanisme terstruktur untuk menangani pengecualian ini dengan menggunakan pernyataan try, catch block, dan akhirnya block.

* A **try block** encloses the code that might throw an exception. Blok percobaan melingkupi kode yang mungkin melemparkan pengecualian.
* A **catch block** handles specific types of exceptions that are thrown within the try block. You can have one or more catch blocks. Blok tangkapan menangani jenis pengecualian tertentu yang dilemparkan di dalam blok percobaan. Anda dapat memiliki satu atau lebih catch block.
* A **finally block** contains cleanup code that *always* executes, regardless of whether an exception was thrown or caught. Blok terakhir berisi kode pembersihan yang selalu dieksekusi, terlepas dari apakah pengecualian dilemparkan atau ditangkap.

A try statement must always be followed by at least one catch block or a finally block, or both. Pernyataan try harus selalu diikuti oleh setidaknya satu blok tangkapan atau blok akhir, atau keduanya.

The basic structure looks like this:

|  |
| --- |
| try {  // Code that might throw an exception } catch (ExceptionA ex) // Catches a specific type of exception (ExceptionA) {  // Handle ExceptionA } catch (ExceptionB ex) // Catches another specific type of exception (ExceptionB) {  // Handle ExceptionB } finally {  // Cleanup code (always executes) } |

### **Example: Handling Division by Zero** **Contoh: Menangani Pembagian dengan Nol**

Consider a simple Calc method that performs division: Pertimbangkan metode Calc sederhana yang melakukan pembagian:

|  |
| --- |
| int Calc(int x) => 10 / x;  // ... // int y = Calc(0); // This would cause a DivideByZeroException and crash the program // Console.WriteLine(y); |

To prevent the program from crashing, we can wrap the call in a try-catch block: Untuk mencegah program mengalami crash, kita dapat membungkus panggilan dalam blok try-catch:

|  |
| --- |
| int Calc(int x) => 10 / x;  // ... try {  int y = Calc(0); // This line will throw DivideByZeroException  Console.WriteLine(y); } catch (DivideByZeroException ex) // Catching the specific exception type {  Console.WriteLine("Error: x cannot be zero.");  // 'ex' contains information about the exception } Console.WriteLine("Program completed."); |

**Output:**

|  |
| --- |
| Error: x cannot be zero. Program completed. |

In this example, instead of terminating, the program catches the DivideByZeroException, prints an error message, and then continues execution. Dalam contoh ini, alih-alih mengakhiri, program menangkap DivideByZeroException, mencetak pesan kesalahan, dan kemudian melanjutkan eksekusi.

**Important Note:** While try-catch is powerful, it's generally preferable to prevent errors with explicit checks when possible. For instance, before calling Calc(x), you could check if (x == 0). Exceptions are relatively expensive in terms of performance, so they should be reserved for truly exceptional and unpredictable situations, not for expected validation. Catatan Penting: Meskipun try-catch sangat kuat, namun secara umum lebih baik mencegah kesalahan dengan pemeriksaan eksplisit jika memungkinkan. Sebagai contoh, sebelum memanggil Calc(x), Anda dapat memeriksa apakah (x == 0). Pengecualian relatif mahal dalam hal performa, jadi pengecualian sebaiknya digunakan untuk situasi yang benar-benar luar biasa dan tidak dapat diprediksi, bukan untuk validasi yang diharapkan.

### **How Exception Handling Works** **Cara Kerja Penanganan Pengecualian**

When an exception is thrown inside a try block: Ketika pengecualian dilemparkan di dalam blok percobaan:

1. The Common Language Runtime (CLR) looks for a compatible catch block immediately following the try block. Common Language Runtime (CLR) mencari blok tangkapan yang kompatibel segera setelah blok percobaan.
   * If a compatible catch block is found, execution jumps to that catch block. After the catch block finishes, the finally block (if present) executes, and then normal program flow resumes. Jika blok tangkapan yang sesuai ditemukan, eksekusi akan melompat ke blok tangkapan tersebut. Setelah blok tangkapan selesai, blok terakhir (jika ada) dieksekusi, lalu aliran program normal dilanjutkan.
   * If no compatible catch block is found, the finally block (if present) executes. Then, the CLR walks up the call stack (to the method that called the current method, and so on) looking for other try blocks with compatible catch handlers. This process repeats until a handler is found. Jika tidak ada blok tangkapan yang kompatibel yang ditemukan, blok terakhir (jika ada) akan dieksekusi. Kemudian, CLR berjalan ke atas tumpukan panggilan (ke metode yang memanggil metode saat ini, dan seterusnya) mencari blok percobaan lain dengan penangan tangkapan yang kompatibel. Proses ini berulang hingga sebuah penangan ditemukan.
2. If no function in the entire call stack takes responsibility for the exception (i.e., no compatible catch block is found anywhere), the program terminates. Jika tidak ada fungsi di seluruh tumpukan panggilan yang bertanggung jawab atas pengecualian (misalnya, tidak ada blok tangkapan yang kompatibel yang ditemukan di mana pun), program akan diakhiri.

## **The catch Clause** **Klausul tangkapan**

A catch clause specifies the type of exception it can handle. This type must be System.Exception or a class derived from it. Klausa tangkapan menentukan jenis pengecualian yang dapat ditangani. Tipe ini harus System.Exception atau kelas yang diturunkan darinya.

* **Catching System.Exception:** This catches *all* possible errors. This is typically done in: Menangkap System.Exception: Ini menangkap semua kesalahan yang mungkin terjadi. Hal ini biasanya dilakukan di:  
  + **Recovery Scenarios:** If your program can genuinely recover regardless of the specific error. Skenario Pemulihan: Jika program Anda benar-benar dapat pulih terlepas dari kesalahan spesifiknya.
  + **Logging and Rethrowing:** To log an error before rethrowing it (perhaps as a different exception type). Pencatatan dan Penarikan Kembali: Untuk mencatat kesalahan sebelum mengembalikannya (mungkin sebagai jenis pengecualian yang berbeda).
  + **Last Resort Handlers:** As a final safety net at the top level of your application to prevent crashes. Penangan Terakhir: Sebagai jaring pengaman terakhir di tingkat teratas aplikasi Anda untuk mencegah kerusakan.
* **Catching Specific Exception Types:** More commonly, you catch specific exception types (e.g., DivideByZeroException, FormatException) to handle known error conditions precisely. This avoids catching unexpected errors that your handler isn't designed for (like OutOfMemoryException). Menangkap Jenis Pengecualian Tertentu: Umumnya, Anda menangkap jenis pengecualian tertentu (misalnya, DivideByZeroException, FormatException) untuk menangani kondisi kesalahan yang diketahui dengan tepat. Hal ini untuk menghindari menangkap kesalahan tak terduga yang tidak dirancang untuk penanganannya (seperti OutOfMemoryException).

You can have **multiple catch clauses** to handle different exception types. The CLR evaluates them in order, from top to bottom. Only **one catch clause executes** for a given exception. Therefore, you must place more specific exception handlers *before* more general ones (e.g., DivideByZeroException before System.Exception). Anda dapat memiliki beberapa klausa penangkap untuk menangani jenis pengecualian yang berbeda. CLR mengevaluasi mereka secara berurutan, dari atas ke bawah. Hanya satu catch clause yang dieksekusi untuk sebuah eksepsi. Oleh karena itu, Anda harus menempatkan penangan eksepsi yang lebih spesifik sebelum eksepsi yang lebih umum (misal, DivideByZeroException sebelum System.Exception).

|  |
| --- |
| static void Main(string[] args) {  try  {  byte b = byte.Parse(args[0]); // args[0] might not exist, or not be a number, or be too large  Console.WriteLine(b);  }  catch (IndexOutOfRangeException) // More specific  {  Console.WriteLine("Please provide at least one argument.");  }  catch (FormatException) // More specific  {  Console.WriteLine("That's not a number!");  }  catch (OverflowException) // More specific  {  Console.WriteLine("You've given me more than a byte!");  }  catch (Exception ex) // General fallback (must be last)  {  Console.WriteLine($"An unexpected error occurred: {ex.Message}");  } } |

You can also catch an exception **without specifying a variable** if you don't need to access its properties: Anda juga dapat menangkap pengecualian tanpa menentukan variabel jika Anda tidak perlu mengakses propertinya:

|  |
| --- |
| catch (OverflowException) { /\* ... \*/ } |

You can even **omit both the variable and the type** to catch all exceptions (this is equivalent to catch (System.Exception) without a variable): Anda bahkan dapat menghilangkan variabel dan tipe untuk menangkap semua pengecualian (ini setara dengan menangkap (System.Exception) tanpa variabel):

|  |
| --- |
| catch { /\* ... \*/ } // Catches all exceptions |

### **Exception Filters (when clause)** **Filter Pengecualian (klausa kapan)**

You can add an **exception filter** to a catch clause using the when keyword. This allows you to catch an exception *only if* a specified boolean condition is true. Anda dapat menambahkan filter pengecualian ke klausa tangkapan menggunakan kata kunci when. Hal ini memungkinkan Anda untuk menangkap pengecualian hanya jika kondisi boolean yang ditentukan benar.

|  |
| --- |
| catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout) {  Console.WriteLine("Web request timed out."); } catch (WebException ex) // This catch will only be considered if the above filter was false {  Console.WriteLine($"Another WebException occurred: {ex.Status}"); } |

If a WebException is thrown, the when condition is evaluated. If it's false, that catch block is skipped, and the CLR continues to look for other compatible catch clauses. The boolean expression in when can even have side effects, such as logging the exception. Jika WebException dilemparkan, kondisi when dievaluasi. Jika bernilai salah, blok tangkapan tersebut akan dilewati, dan CLR akan terus mencari klausa tangkapan lain yang kompatibel. Ekspresi boolean di dalam when bahkan dapat memiliki efek samping, seperti mencatat pengecualian.

## **The finally Block**

A finally block guarantees that its code will execute, irrespective of how the try block is exited. This makes it ideal for **cleanup tasks** that must always happen, such as closing file handles, database connections, or releasing resources. Blok terakhir menjamin bahwa kodenya akan dieksekusi, terlepas dari bagaimana blok percobaan keluar. Hal ini membuatnya ideal untuk tugas pembersihan yang harus selalu dilakukan, seperti menutup pegangan berkas, koneksi basis data, atau melepaskan sumber daya.

A finally block executes in the following scenarios: Blok terakhir dieksekusi dalam skenario berikut:

* After a catch block finishes (or throws a new exception). Setelah blok tangkapan selesai (atau melempar pengecualian baru).
* After the try block finishes normally. Setelah blok percobaan selesai secara normal.
* After the try block throws an exception for which there is no compatible catch block. Setelah blok percobaan melempar pengecualian yang tidak ada blok tangkapan yang kompatibel.
* If control leaves the try block due to a jump statement (e.g., return, break, goto). Jika kontrol meninggalkan blok percobaan karena pernyataan lompatan (misal: return, break, goto).

The only situations that can prevent a finally block from executing are an infinite loop within the try or catch block, or the process being abruptly terminated (e.g., by the operating system). Satu-satunya situasi yang dapat mencegah blok terakhir dieksekusi adalah perulangan tak terbatas di dalam blok try atau catch, atau proses dihentikan secara tiba-tiba (mis., oleh sistem operasi).

**Example: Ensuring a File is Closed**

|  |
| --- |
| void ReadFile() {  StreamReader reader = null; // Declare outside try for finally block access  try  {  reader = File.OpenText("file.txt");  if (reader.EndOfStream) return; // Returns early, but finally still executes  Console.WriteLine(reader.ReadToEnd());  }  finally  {  if (reader != null)  reader.Dispose(); // Ensures the file stream is closed  } } |

In this example, reader.Dispose() is called even if an IOException occurs during ReadToEnd() or if the method returns early. Dalam contoh ini, reader.Dispose() dipanggil meskipun terjadi IOException selama ReadToEnd() atau jika metode dikembalikan lebih awal.

### **The using Statement and IDisposable** **Pernyataan penggunaan dan IDisposable**

Many classes that manage unmanaged resources (like file handles, network connections) implement the System.IDisposable interface, which has a single method: Dispose(). The using statement in C# provides a syntactic sugar for ensuring Dispose() is called on such objects within a finally block. Banyak kelas yang mengelola sumber daya yang tidak dikelola (seperti pegangan file, koneksi jaringan) mengimplementasikan antarmuka System.IDisposable, yang memiliki metode tunggal: Buang(). Pernyataan menggunakan dalam C# menyediakan sintaksis untuk memastikan Dispose() dipanggil pada objek-objek seperti itu di dalam blok akhirnya.

|  |
| --- |
| // The using statement: using (StreamReader reader = File.OpenText("file.txt")) // Reader is automatically disposed {  // Use reader here  Console.WriteLine(reader.ReadToEnd()); } // At the end of this block, reader.Dispose() is implicitly called |

This is precisely equivalent to the try-finally block shown above, making code cleaner and safer. Hal ini sama persis dengan blok try-finally yang ditunjukkan di atas, sehingga membuat kode menjadi lebih bersih dan aman.

### **using Declarations (C# 8+)**

If you omit the curly braces and statement block after a using statement, it becomes a **using declaration**. The resource will be disposed when execution leaves the *enclosing scope* (the block in which the using declaration is made). Jika Anda menghilangkan tanda kurung kurawal dan blok pernyataan setelah pernyataan menggunakan, maka akan menjadi deklarasi menggunakan. Sumber daya akan dibuang ketika eksekusi meninggalkan ruang lingkup yang diapit (blok di mana deklarasi penggunaan dibuat).

|  |
| --- |
| if (File.Exists("file.txt")) {  using var reader = File.OpenText("file.txt"); // reader will be disposed when leaving the 'if' block  Console.WriteLine(reader.ReadLine());  // ... more code using reader } // reader.Dispose() is called here |

## **Throwing Exceptions**

Exceptions can be thrown by the CLR (as seen with DivideByZeroException) or explicitly by your own code using the throw keyword. Pengecualian dapat dilemparkan oleh CLR (seperti yang terlihat pada DivideByZeroException) atau secara eksplisit oleh kode Anda sendiri dengan menggunakan kata kunci throw.

|  |
| --- |
| void Display(string name) {  if (name == null)  throw new ArgumentNullException(nameof(name), "Name cannot be null."); // Throw an exception  Console.WriteLine(name); }  // ... try { Display(null); } catch (ArgumentNullException ex) {  Console.WriteLine($"Caught the exception: {ex.Message}"); } |

**ArgumentNullException.ThrowIfNull (from .NET 6):** For common null checks, a concise helper exists: ArgumentNullException.ThrowIfNull (dari .NET 6): Untuk pemeriksaan null yang umum, tersedia sebuah pembantu yang ringkas:

|  |
| --- |
| void Display(string name) {  ArgumentNullException.ThrowIfNull(name); // Throws ArgumentNullException if name is null  Console.WriteLine(name); } |

### **throw Expressions (C# 7+)**

The throw keyword can also be used as an expression, allowing it in places like expression-bodied members or ternary conditional expressions: Kata kunci throw juga dapat digunakan sebagai ekspresi, memungkinkannya digunakan di tempat-tempat seperti anggota bertubuh ekspresi atau ekspresi kondisional terner:

|  |
| --- |
| public string Foo() => throw new NotImplementedException(); // Throw as an expression  string ProperCase(string value) =>  value == null ? throw new ArgumentException("Value cannot be null.") : // Throw in ternary  value == "" ? "" :  char.ToUpper(value[0]) + value.Substring(1); |

### **Rethrowing an Exception**

You can catch an exception, perform some actions (like logging), and then **rethrow** the *original* exception. Anda dapat menangkap pengecualian, melakukan beberapa tindakan (seperti pencatatan), dan kemudian mengembalikan pengecualian asli.

|  |
| --- |
| try {  // Some risky operation } catch (Exception ex) {  // Log the error: ex.Message, ex.StackTrace, etc.  Console.WriteLine($"Logged error: {ex.Message}");  throw; // Rethrows the ORIGINAL exception, preserving its stack trace } |

**Important:** Using throw; (without ex) is crucial for preserving the original stack trace of the exception. If you use throw ex;, the stack trace will reset to the point where throw ex; was called, losing valuable debugging information about the original source of the error. Penting: Menggunakan throw; (tanpa ex) sangat penting untuk mempertahankan jejak stack asli dari eksepsi. Jika Anda menggunakan throw ex;, stack trace akan di-reset ke titik di mana throw ex; dipanggil, sehingga kehilangan informasi debug yang berharga mengenai sumber asli kesalahan.

You can also rethrow a **new, more specific exception** while wrapping the original one: Anda juga dapat membuat ulang pengecualian baru yang lebih spesifik sambil membungkus pengecualian yang asli:

|  |
| --- |
| try {  // Parse a DateTime from XML element data } catch (FormatException ex) {  // Wrap the original exception for debugging purposes  throw new XmlException("Invalid DateTime format in XML.", ex); } |

Passing the original ex to the new exception's constructor populates the InnerException property, forming a chain of exceptions that helps trace the root cause. This is a best practice. Meneruskan eks asli ke konstruktor pengecualian baru akan mengisi properti InnerException, membentuk rantai pengecualian yang membantu melacak akar masalah. Ini adalah praktik terbaik.

## **Key Properties of System.Exception**

All exceptions in C# derive from System.Exception. Key properties for debugging and handling include: Semua pengecualian dalam C# berasal dari System.Exception. Properti utama untuk debugging dan penanganan meliputi:

* **StackTrace**: A string showing the sequence of method calls leading up to where the exception was thrown. Essential for debugging. StackTrace: String yang menunjukkan urutan pemanggilan metode yang mengarah ke tempat pengecualian dilemparkan. Sangat penting untuk debugging.
* **Message**: A string describing the error in detail. Pesan: String yang menjelaskan kesalahan secara detail.
* **InnerException**: If an exception was rethrown and wrapped another exception, this property holds the original, inner exception. This can form a chain of exceptions. InnerException: Jika sebuah pengecualian dibatalkan dan membungkus pengecualian lain, properti ini menyimpan pengecualian asli, pengecualian dalam. Hal ini dapat membentuk sebuah rantai pengecualian.

**Note:** C# has only **runtime exceptions** (unchecked exceptions). There is no equivalent to Java's compile-time "checked exceptions" that force you to declare what exceptions a method might throw. Catatan: C# hanya memiliki pengecualian runtime (pengecualian yang tidak dicentang). Tidak ada padanan untuk “pengecualian yang dicentang” pada waktu kompilasi Java yang memaksa Anda untuk mendeklarasikan pengecualian apa yang mungkin dilemparkan oleh sebuah metode.

## **Common Exception Types** **Jenis Pengecualian Umum**

You'll frequently encounter and potentially throw these exception types: Anda akan sering menemukan dan berpotensi melempar jenis pengecualian ini:

* **System.ArgumentException**: General exception for an invalid argument passed to a method. Usually indicates a bug in the calling code. System.ArgumentException: Pengecualian umum untuk argumen yang tidak valid yang diteruskan ke metode. Biasanya mengindikasikan adanya kesalahan dalam kode pemanggilan.
* **System.ArgumentNullException**: A subclass of ArgumentException, specifically for null arguments where null is not allowed. System.ArgumentNullException: Subkelas dari ArgumentException, khusus untuk argumen null di mana null tidak diperbolehkan.
* **System.ArgumentOutOfRangeException**: Another subclass of ArgumentException, for when an argument (often numeric) is outside an acceptable range. System.ArgumentOutOfRangeException: Subkelas lain dari ArgumentException, untuk saat argumen (biasanya berupa angka) berada di luar rentang yang dapat diterima.
* **System.InvalidOperationException**: Thrown when an object's state is invalid for a particular operation (e.g., trying to read from a closed file). The arguments might be valid, but the object itself isn't ready. System.InvalidOperationException: Dilemparkan ketika status objek tidak valid untuk operasi tertentu (misalnya, mencoba membaca dari file yang tertutup). Argumennya mungkin valid, tetapi objeknya sendiri tidak siap.
* **System.NotSupportedException**: Indicates that a particular operation or feature is not implemented or supported (e.g., calling Add on a read-only collection). System.NotSupportedException: Menunjukkan bahwa operasi atau fitur tertentu tidak diimplementasikan atau didukung (misalnya, memanggil Tambah pada koleksi hanya-baca).
* **System.NotImplementedException**: Indicates that a method or property has not yet been implemented (often used as a placeholder in development). System.NotImplementedException: Mengindikasikan bahwa sebuah metode atau properti belum diimplementasikan (sering digunakan sebagai placeholder dalam pengembangan).
* **System.ObjectDisposedException**: Thrown when you try to use an object that has already been disposed. System.ObjectDisposedException: Dilemparkan ketika Anda mencoba menggunakan objek yang telah dibuang.
* **System.NullReferenceException**: Thrown by the CLR when you attempt to access a member (field, property, method) of an object whose value is null. This nearly always indicates a programming error. System.NullReferenceException: Dilemparkan oleh CLR ketika Anda mencoba mengakses anggota (bidang, properti, metode) dari sebuah objek yang nilainya nol. Hal ini hampir selalu mengindikasikan kesalahan pemrograman.

## **The TryXXX Method Pattern**

For operations where failure is a common or expected outcome (rather than an exceptional one), it's often useful to offer two versions of a method: Untuk operasi di mana kegagalan adalah hasil yang umum atau yang diharapkan (bukan hasil yang luar biasa), sering kali berguna untuk menawarkan dua versi metode:

* A **throwing method** (e.g., Parse): Throws an exception on failure. Metode pelemparan (misalnya, Parse): Melemparkan pengecualian pada kegagalan.
* A **TryXXX method** (e.g., TryParse): Returns a bool indicating success/failure and an out parameter for the result. Sebuah metode TryXXX (misalnya, TryParse): Mengembalikan sebuah bool yang mengindikasikan keberhasilan/kegagalan dan sebuah parameter keluar untuk hasilnya.

**Example: int.Parse vs. int.TryParse**

|  |
| --- |
| // Throws an exception if parsing fails int number1 = int.Parse("123"); // OK // int number2 = int.Parse("abc"); // Throws FormatException  // Returns true/false and provides result via 'out' parameter if (int.TryParse("123", out int result1)) {  Console.WriteLine($"Parsed: {result1}"); // Output: Parsed: 123 }  if (!int.TryParse("abc", out int result2)) {  Console.WriteLine("Failed to parse 'abc'."); // Output: Failed to parse 'abc'. } |

This pattern is ideal when failures are part of the normal program flow and can be handled without resorting to try-catch blocks, which are better for truly exceptional situations. Pola ini sangat ideal ketika kegagalan merupakan bagian dari alur program normal dan dapat ditangani tanpa harus menggunakan blok try-catch, yang lebih baik untuk situasi yang benar-benar luar biasa.