

PANDUAN LENGKAP PEMOGRAMAN

C#

KATA PENGANTAR

Puji syukur penulis panjatkan ke hadirat Tuhan Yang Maha Esa atas segala rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan penyusunan e-book ini dengan baik. E-book ini dibuat sebagai sarana pembelajaran untuk memberikan pemahaman dasar mengenai bahasa pemrograman C#, khususnya bagi pemula yang ingin mengenal dan mempelajari dunia pemrograman.

Materi yang disajikan dalam e-book ini meliputi pengenalan bahasa C#, dasar-dasar sintaks, struktur program, serta contoh penerapan sederhana yang mudah dipahami. Penulis berusaha menyusun materi secara runtut dan sistematis dengan menggunakan bahasa yang sederhana agar dapat membantu pembaca dalam memahami isi yang disampaikan.

E-book ini ditujukan terutama bagi siswa Sekolah Menengah Kejuruan (SMK), khususnya pada bidang Pengembangan Perangkat Lunak dan Gim (PPLG). Penulis menyadari bahwa penyusunan e-book ini masih memiliki kekurangan. Oleh karena itu, penulis sangat mengharapkan kritik dan saran yang membangun demi perbaikan di masa yang akan datang.

Akhir kata, penulis berharap semoga e-book ini dapat memberikan manfaat, menambah pengetahuan, serta membantu pembaca dalam memahami dasar-dasar pemrograman C# dengan lebih baik.

TENTANG PENULIS

Nabila Syadsa adalah siswi SMK Negeri 1 Bawang yang sedang menempuh pendidikan di kelas X PPLG 2 (Pengembangan Perangkat Lunak dan Gim). Ia memiliki ketertarikan pada bidang teknologi informasi, khususnya dalam dunia pemrograman. Ketertarikan tersebut membuat penulis mulai mempelajari berbagai bahasa pemrograman dasar, termasuk C#, sebagai langkah awal untuk memahami konsep coding secara lebih mendalam.

Melalui penyusunan e-book ini, penulis berupaya menuangkan hasil pembelajaran yang telah diperoleh agar dapat bermanfaat bagi pembaca, terutama bagi pemula. Penulis berharap materi yang disajikan dapat membantu dalam memahami dasar-dasar pemrograman secara lebih mudah dan terarah, serta menjadi motivasi untuk terus belajar dan mengembangkan kemampuan di bidang teknologi.

CARA MENGGUNAKAN BUKU INI

Agar pembaca dapat memperoleh hasil belajar yang optimal dari buku ini, disarankan untuk mengikuti beberapa langkah berikut:

1. Pelajari materi secara bertahap mulai dari bagian awal hingga selesai.
2. Pastikan memahami konsep dasar sebelum melanjutkan ke pembahasan selanjutnya.
3. Amati dengan baik setiap contoh kode yang disediakan.
4. Cobalah langsung praktikkan kode program menggunakan perangkat komputer.
5. Lakukan pengulangan latihan apabila masih mengalami kesulitan dalam memahami materi.
6. Manfaatkan buku ini sebagai referensi belajar, baik secara mandiri maupun dengan bimbingan guru.

Buku ini disusun dengan bahasa yang sederhana agar mudah dipahami oleh pembaca pemula, sehingga sangat sesuai bagi siswa yang baru mulai belajar pemrograman.

PERSYARATAN MEMBACA

Sebelum mulai mempelajari isi buku ini, pembaca disarankan untuk memiliki beberapa hal berikut:

1. Pengetahuan dasar mengenai cara menggunakan komputer.
2. Kemampuan dasar dalam mengoperasikan sistem operasi seperti Windows atau Linux.
3. Ketertarikan serta motivasi untuk mempelajari dunia pemrograman.
4. Perangkat komputer atau laptop yang dapat digunakan untuk praktik secara langsung.
5. Perangkat lunak pendukung, antara lain:
 - Visual Studio atau Visual Studio Code
 - .NET SDK

Buku ini dirancang untuk pemula, sehingga tidak mengharuskan pembaca memiliki pengalaman sebelumnya dalam bidang pemrograman karena materi dimulai dari tahap dasar.

Bawang, 2026

Penulis

DAFTAR ISI

KATA PENGANTAR	2
TENTANG PENULIS	3
CARA MENGGUNAKAN BUKU INI	4
PERSYARATAN MEMBACA	5
DAFTAR ISI	6
BAGIAN 1: DASAR-DASAR C#	7
Bab 1: Pengenalan C# dan .NET Framework	7
Bab 2: Sintaks Dasar C#	12
Bab 3: Output dan Komentar	15
1. Console.WriteLine()	16
Bab 4: Variabel dan Konstanta.....	19
Bab 5: Tipe Data dalam C#	25
Bab 6: Type Casting dan Konversi	31
Bab 7: Input dari Pengguna	37
Bab 8: Operator dalam C#	41
Bab 9: Kelas Math dan Fungsi Matematika	51
BAGIAN 2: STRUKTUR KONTROL DAN STRING	63
Bab 10: Manipulasi String	63
Bab 11: Boolean dan Logika Kondisional	76
Bab 12: Percabangan: If, Else, dan Switch	81
Bab 13: Perulangan (Looping)	92
A. Pernyataan break	95
Bab 14: Array dan Koleksi Dasar	103
BAGIAN 3: METODE DAN PEMROGRAMAN TERSTRUKTUR	112
Bab 15: Metode (Fungsi) dalam C#	112
Value vs Reference (ref, out)	125
BAGIAN 4: PEMROGRAMAN BERORIENTASI OBJEK (OOP)	136
Bab 17: Pengenalan OOP dalam C#	136
Bab 18: Class dan Object	139
Bab 19: Constructor dan Destructor	145

BAGIAN 1: DASAR-DASAR C#

Bab 1: Pengenalan C# dan .NET Framework

- 1.1 Sejarah dan Perkembangan C#

C# (dibaca: **C-Sharp**) adalah bahasa pemrograman modern, berorientasi objek, dan bersifat *type-safe* yang dikembangkan oleh Microsoft. Sejak kemunculannya, C# telah bertransformasi dari sekadar "pesaing Java" menjadi salah satu bahasa paling serbaguna di dunia.

1. Asal-Usul dan Kelahiran (Akhir 1990-an - 2002)

Pada akhir 1990-an, Microsoft ingin memiliki bahasa pemrograman yang terintegrasi penuh dengan ekosistem Windows dan platform .NET yang saat itu sedang dikembangkan.

- **Proyek Cool:** Awalnya, bahasa ini diberi nama kode "**Cool**" (*C-like Object Oriented Language*).
- **Tokoh Kunci: Anders Hejlsberg**, arsitek terkenal di balik Turbo Pascal dan Delphi, ditunjuk untuk memimpin tim pengembangan.
- **Perilisan:** Pada Juli 2000, di konferensi Professional Developers Conference (PDC), Microsoft mengumumkan C#. Versi 1.0 resmi dirilis pada tahun **2002** bersamaan dengan Visual Studio .NET 2002.

2. Evolusi Versi C#

C# terus berkembang secara konsisten dengan menambahkan fitur-fitur yang memudahkan pengembang. Berikut adalah tonggak sejarah utamanya:

Era Awal (C# 1.0 & 2.0)

- **C# 1.0 (2002):** Dasar-dasar bahasa pemrograman berorientasi objek (OOP).
- **C# 2.0 (2005):** Memperkenalkan fitur revolusioner seperti **Generics**, *Partial Types*, dan *Anonymous Methods*. Ini membuat kode menjadi lebih fleksibel dan efisien.

Era Inovasi (C# 3.0 - 5.0)

- **C# 3.0 (2007):** Merupakan lompatan besar dengan diperkenalkannya **LINQ** (*Language Integrated Query*), *Lambda Expressions*, dan *Extension Methods*. C# mulai mengadopsi gaya pemrograman fungsional.
- **C# 4.0 (2010):** Fokus pada interoperabilitas dengan pengenalan kata kunci *dynamic*.

- **C# 5.0 (2012):** Merevolusi pemrograman asinkronus dengan fitur **Async/Await**, memudahkan pengembang menangani operasi berat tanpa memblokir antarmuka pengguna.

Era Modern dan Open Source (C# 6.0 - Sekarang)

- **C# 6.0 & 7.0:** Memperkenalkan banyak "pemanis sintaksis" (*syntactic sugar*) seperti *String Interpolation*, *Pattern Matching*, dan *Tuples*.
- **C# 8.0 - 12.0:** Fokus pada keamanan data (seperti *Nullable Reference Types*) dan performa tinggi. C# kini mendukung pengembangan lintas platform (*Cross-platform*) secara penuh melalui **.NET Core** (sekarang hanya disebut **.NET**).

3. Mengapa C# Menjadi Populer?

Perkembangan C# tidak lepas dari kemampuannya beradaptasi dengan tren industri. Beberapa faktor kunci meliputi:

1. **Dukungan Multi-Platform:** Melalui .NET (sebelumnya .NET Core), C# sekarang bisa berjalan di Windows, Linux, dan macOS.
2. **Ekosistem Game (Unity):** C# adalah bahasa utama yang digunakan dalam engine game **Unity**, menjadikannya standar industri bagi pengembang game indie maupun AAA.
3. **Pengembangan Mobile:** Dengan **Xamarin/MAUI**, pengembang dapat membangun aplikasi Android dan iOS menggunakan basis kode C#.
4. **Keamanan dan Performa:** C# menggabungkan produktivitas tinggi (seperti Python) dengan performa yang mendekati C++.

4. Kondisi C# Saat Ini

Saat ini, C# bukan lagi sekadar bahasa untuk aplikasi desktop Windows. C# digunakan secara luas untuk:

- **Web Development:** Menggunakan ASP.NET Core yang sangat cepat dan skalabel.
- **Cloud Computing:** Integrasi mendalam dengan layanan Microsoft Azure.
- **Artificial Intelligence:** Melalui pustaka seperti ML.NET.

Dengan jadwal rilis tahunan yang stabil dari Microsoft, C# terus mempertahankan posisinya sebagai salah satu bahasa pemrograman paling dicintai dan paling banyak digunakan di dunia (sering menduduki peringkat atas di survei Stack Overflow dan TIOBE Index).

- **1.2 Apa itu .NET Framework?**

.NET Framework adalah kerangka kerja (framework) buatan Microsoft untuk membangun dan menjalankan aplikasi, terutama di sistem operasi Microsoft Windows.

Fungsi .NET Framework:

1. **Membuat aplikasi** seperti aplikasi desktop, web, dan game.
2. **Menyediakan library** (kumpulan kode siap pakai) untuk mempermudah pemrograman.
3. **Menjalankan program** melalui **Common Language Runtime (CLR)**, yang mengatur eksekusi program, memori, dan keamanan.
4. **Mendukung banyak bahasa pemrograman**, seperti **C#, VB.NET**, dan **F#**.

Komponen utama .NET Framework:

- **CLR (Common Language Runtime)** → mesin yang menjalankan program.
- **Framework Class Library (FCL)** → kumpulan fungsi dan class siap pakai.

Contoh penggunaan:

- Membuat aplikasi kasir
 - Website berbasis **ASP.NET**
 - Aplikasi desktop dengan **Windows Forms** atau **WPF**
- Singkatnya, **.NET Framework adalah platform untuk membuat, menjalankan, dan mengelola aplikasi berbasis Windows.**

Kalau mau penjelasan versi singkat untuk tugas sekolah juga bisa aku bantu.

• **1.3 Keunggulan dan Kegunaan C#**

Keunggulan C#

1. Mudah dipelajari karena sintaksnya sederhana dan terstruktur.
2. Berorientasi objek (OOP) sehingga memudahkan pembuatan program yang rapi dan terorganisir.
3. Cepat dan efisien untuk mengembangkan aplikasi.
4. Keamanan tinggi karena memiliki fitur pengelolaan memori dan keamanan bawaan.
5. Banyak library/framework yang membantu mempercepat pembuatan program.
6. Multiplatform dapat digunakan di berbagai sistem operasi melalui .NET.
7. Didukung Microsoft sehingga terus dikembangkan.

Kegunaan C#

1. Membuat aplikasi desktop (misalnya dengan **Windows Forms** atau **WPF**).
2. Membuat website dan aplikasi web menggunakan **ASP.NET**.
3. Mengembangkan game, terutama dengan Unity.
4. Membuat aplikasi mobile dengan **Xamarin** / **.NET MAUI**.
5. Membuat aplikasi bisnis atau database.
6. Digunakan untuk pengembangan AI, IoT, dan cloud computing.

• **1.4 Menyiapkan Environment: Visual Studio / VS Code / .NET SDK**

Menyiapkan Environment untuk C#

1. Menggunakan Visual Studio

Langkah-langkah:

- Download dan install **Visual Studio**.
- Saat instalasi pilih workload **.NET Desktop Development** atau **ASP.NET and Web Development** sesuai kebutuhan.
- Setelah terpasang, buat project baru (**Create a new project**).

- Pilih project **Console App (C#)** untuk latihan dasar.
Kelebihan:
- Fitur lengkap (debugging, desain GUI, compiler bawaan).
- Cocok untuk pemula dan proyek besar.

2. Menggunakan Visual Studio Code

Langkah-langkah:

- Install **VS Code**.
- Install extension **C#**.
- Install **.NET SDK**.
- Buka terminal lalu buat project dengan perintah:
dotnet new console
dotnet run

Kelebihan:

- Ringan dan cepat.
- Cocok untuk coding sederhana dan latihan.

3. Install .NET SDK Fungsi .NET SDK:

- Untuk menulis, compile, dan menjalankan program C#.

Cek instalasi:

dotnet --version

Jika muncul nomor versi, berarti berhasil.

Perintah dasar:

dotnet new console # membuat project baru

dotnet run # menjalankan program

dotnet build # compile program

• 1.5 Program "Hello World" Pertama Anda

Penjelasan:

- **using System;** Menggunakan library **System** agar bisa memakai Console.
- **class Program:** Membuat kelas bernama **Program**.
- **static void Main():** Method utama, tempat program mulai dijalankan.
- **Console.WriteLine("Hello World!");** Menampilkan tulisan **Hello World!** ke layar.

```

1  using System;
2
3  class Program
4  {
5      static void Main()
6      {
7          Console.WriteLine("Hello World!");
8      }
9  }

```

Cara Menjalankan

Jika memakai Visual Studio:

- Buat project **Console App**
- Ganti kode dengan program di atas
- Tekan **Run / F5**

• 1.6 Memahami Struktur Dasar Program C#

```

1  using System;
2
3  namespace BelajarCSharp
4  {
5      0 references
6      class Program
7      {
8          0 references
9          static void Main(string[] args)
10         {
11             Console.WriteLine("Halo Dunia!");
12         }
13     }
14 }

```

Struktur Dasarnya:

1. using System;

Digunakan untuk memanggil library **System** agar bisa memakai fungsi seperti Console.WriteLine().

2. namespace BelajarCSharp

Namespace berfungsi mengelompokkan kode agar lebih terorganisir.

3. class Program

Class adalah wadah untuk menyimpan program dan method.

4. static void Main(string[] args)

Ini adalah **method utama** (titik awal program berjalan).

- static → bisa dijalankan tanpa membuat objek.
- void → tidak mengembalikan nilai.
- Main() → fungsi utama program.
- string[] args → menerima input dari command line.

5. Console.WriteLine("Halo Dunia!");

Perintah untuk menampilkan teks ke layar.

Struktur Sederhana Program C#

Urutannya biasanya:

Using directives

Namespace

Class

Main method

Statements (perintah)

```

1  using System;
2
3  namespace
4  {
5      class
6      {
7          0 references
8          static void Main()
9          {
10             // kode program
11          }
12     }

```

Bab 2: Sintaks Dasar C#

• 2.1 Aturan Penulisan Kode C#

1. Setiap pernyataan diakhiri titik koma (;)

```

1  int umur = 17;
2  Console.WriteLine(umur);

```

2. C# bersifat case sensitive: Huruf besar dan kecil dibedakan.
Benar:

```

1  int nilai = 90;

```

3. Gunakan kurung kurawal { }

```

1  if (nilai > 75)
2  {
3      Console.WriteLine("Lulus");
4  }

```

4. Nama variabel tidak boleh sembarangan
 - Diawali huruf atau underscore (_)
 - Mengandung angka (bukan diawal)

```

1  string namaSiswa;
2  int nilail;

```

5. Tidak boleh memakai kata kunci (keyword) sebagai nama variabel
Contoh keyword: **int**, **class**, **if**, **while**

6. Gunakan PascalCase dan camelCase

-PascalCase → untuk class/method
Contoh: NamaSiswa, HitungNilai()

-camelCase → untuk variabel
Contoh: namaSiswa, totalNilai

• 2.2 Statement dan Blok Kode

Statement adalah instruksi tunggal yang diberikan kepada komputer untuk melakukan tindakan tertentu. Di C#, setiap statement **wajib** diakhiri dengan tanda titik koma (;). Tanpa tanda ini, program tidak akan bisa berjalan (*compile error*). Bayangkan statement seperti sebuah kalimat dalam bahasa manusia yang harus diakhiri dengan tanda titik.

Contoh Statement:

- **Deklarasi variabel:** `int angka = 10;`
- **Pemanggilan fungsi:** `Console.WriteLine("Halo!");`
- **Operasi matematika:** `hasil = x + y;`

Blok Kode adalah kumpulan dari satu atau lebih statement yang dikelompokkan bersama. Blok kode ditandai dengan kurung kurawal buka ({) dan kurung kurawal tutup (}).

Tujuan utama blok kode adalah untuk memberi tahu komputer bahwa semua instruksi di dalamnya adalah satu kesatuan, biasanya digunakan pada struktur kendali (seperti if, for) atau fungsi.

Contoh Struktur Blok Kode:

```
1 {  
2     // Ini adalah awal blok kode  
3     Console.WriteLine("Baris ini di dalam blok.");  
4     Console.WriteLine("Baris ini juga di dalam blok.");  
5     // Ini adalah akhir blok kode  
6 }
```

3. Hubungan Statement dan Blok Kode

Blok kode sering kali membungkus banyak statement untuk dijalankan dalam kondisi tertentu. Perhatikan contoh di bawah ini:

Aturan Penting:

```
1 if (true)  
2 { // Mulai Blok Kode  
3     string pesan = "Kondisi benar!"; // Statement 1  
4     Console.WriteLine(pesan);       // Statement 2  
5 } // Akhir Blok Kode
```

1. **Semicolon (;):** Digunakan setelah statement, tetapi **tidak** digunakan setelah kurung kurawal tutup (}) pada blok kode (kecuali pada kasus tertentu seperti *object initializer*).
2. **Scope (Cakupan):** Variabel yang dibuat di dalam sebuah blok kode biasanya tidak bisa diakses dari luar blok tersebut. Ini disebut dengan konsep *Local Scope*.

4. Indentasi (Kerapian Kode)

Meskipun C# tidak mewajibkan indentasi (spasi di awal baris) agar kode bisa berjalan, indentasi sangat krusial untuk keterbacaan.

Buruk (Sulit dibaca):

```
1 if (true)  
2 { // Mulai Blok Kode  
3     string pesan = "Kondisi benar!"; // Statement 1  
4     Console.WriteLine(pesan);       // Statement 2  
5 } // Akhir Blok Kode
```

2.3 Whitespace dan Formatting

Whitespace merujuk pada karakter yang tidak terlihat di layar, seperti:

- **Spasi** (Spacebar)
- **Tab**
- **Baris Baru** (Enter/Newline)

Dalam C#, *compiler* umumnya mengabaikan *whitespace* tambahan. Artinya, kedua kode di bawah ini hasilnya **sama persis** bagi komputer:

Versi Padat (Sulit dibaca):

```
1  int x = 10; if (x > 5) { Console.WriteLine("Besar"); }
```

2. Aturan Umum Formatting di C#

Untuk menjaga konsistensi, komunitas C# (dan Microsoft) memiliki standar yang diikuti hampir semua pengembang di seluruh dunia:

A. Peletakan Kurung Kurawal (Allman Style)

Di C#, kurung kurawal buka { biasanya diletakkan di **baris baru**, bukan di akhir statement if atau method.

- **Benar:**

```
1  void Halo()
2  {
3      // Kode di sini
4  }
```

B. Indentasi (4 Spasi/1 Tab)

Setiap kali Anda masuk ke dalam blok kode { }, Anda harus memberikan jarak ke dalam. Ini membantu mata kita melihat hirarki atau "tingkatan" kode dengan cepat.

C. Spasi di Sekitar Operator

Gunakan spasi di sekitar operator matematika atau penugasan agar kode tidak terlihat "sesak".

- `int hasil = a + b;` (Lebih baik)
- `int hasil=a+b;` (Kurang rapi)

3. Mengapa Formatting Itu Penting?

1. **Debugging Lebih Cepat:** Kode yang rapi memudahkan Anda menemukan di mana sebuah blok dimulai dan berakhir.
2. **Kerja Tim:** Jika Anda bekerja dalam tim, semua orang harus mengikuti standar yang sama agar kode terlihat seolah-olah ditulis oleh satu orang.
3. **Mencegah Bug:** Salah meletakkan titik koma atau kurung kurawal akibat kode yang berantakan adalah penyebab umum program *error*.

4. Tips Cepat: Auto-Format!

Kabar baiknya, Anda tidak perlu merapikan kode secara manual satu per satu. Editor modern punya fitur otomatis:

- **Visual Studio:** Tekan Ctrl + K lalu Ctrl + D.
- **VS Code:** Tekan Shift + Alt + F (Windows) atau Shift + Option + F (Mac).
Editor akan langsung "Menyulap" kode berantakan Anda menjadi sangat rapi sesuai standar C#.

2.4 Konvensi Penamaan (Naming Conventions)

1. PascalCase vs camelCase

Sebelum masuk ke detail, mari kita bedakan kedua istilah ini:

2. PascalCase: Huruf pertama dari setiap kata adalah huruf kapital.

Contoh: NamaLengkap, HitungTotalGaji, PemainBaru.

3. **camelCase:** Huruf pertama dari kata pertama adalah huruf kecil, dan huruf pertama dari kata-kata selanjutnya adalah huruf kapital.
Contoh: namaLengkap, hitungTotalGaji, pemainBaru.
4. **Aturan Tambahan yang Penting**
Selain gaya tulisan, ada beberapa aturan tak tertulis (tapi sangat disarankan) dalam memilih kata:
 - A. **Gunakan Nama yang Deskriptif**
Hindari nama variabel satu huruf (kecuali untuk indeks dalam *looping* seperti i atau j).
Buruk: int d = 30;
Baik: int jumlahHariDalamBulan = 30;
 - B. **Kata Kerja vs Kata Benda**
Class & Variabel: Gunakan **Kata Benda** karena mereka mewakili sebuah "objek" atau "benda". (Contoh: User, HargaProduk).
Method: Gunakan **Kata Kerja** karena mereka mewakili sebuah "aksi" atau "tindakan". (Contoh: AmbilData(), HapusUser()).
 - C. **Hindari Singkatan yang Membingungkan**
Jangan menyingkat kata jika itu bisa menimbulkan multitafsir bagi orang lain yang membaca kode Anda.
Buruk: GetAddr()
Baik: GetAddress()

1. **Keterbacaan:** Anda (atau rekan tim) bisa langsung tahu hanya dengan melihat namanya apakah itu sebuah Class, Method, atau hanya variabel lokal.
2. **Integrasi Library:** Karena semua pustaka (*library*) .NET menggunakan standar ini, kode Anda akan terlihat menyatu secara alami dengan kode bawaan Microsoft.
3. **Mencegah Konflik:** Menghindari kebingungan antara nama tipe (Class) dengan nama variabel.

Contoh Gabungan:

```

1 namespace ManajemenSekolah // Namespace (PascalCase)
2 {
3     0 references
4     public class Siswa // Class (PascalCase)
5     {
6         private string _idSiswa; // Private Field (underscore + camelCase)
7
8         0 references
9         public string NamaSiswa { get; set; } // Property (PascalCase)
10
11         0 references
12         public void UpdateData(string namaBaru) // Method (PascalCase) & Parameter (camelCase)
13         {
14             string pesanLog = "Mengubah data..."; // Local Variable (camelCase)
15             Console.WriteLine(pesanLog);
16         }
17     }
18 }

```

Bab 3: Output dan Komentar

- 3.1 Menampilkan Output dengan `Console.WriteLine()` dan `Console.Write()`

1. Console.WriteLine()

WriteLine adalah perintah yang paling sering digunakan. Fungsinya adalah menampilkan teks ke layar dan **secara otomatis berpindah ke baris baru** setelah teks tersebut dicetak.

Contoh:

```
1 Console.WriteLine("Halo");
2 Console.WriteLine("Dunia");
```

2. Console.Write()

Write berfungsi menampilkan teks ke layar, tetapi **tetap berada di baris yang sama**. Teks berikutnya yang Anda cetak akan muncul tepat di sebelah teks sebelumnya.

Contoh:

```
1 Console.Write("Halo ");
2 Console.Write("Dunia");
```

- 3.2 Escape Sequences dan Formatting Output

1. Escape Sequences

Escape sequence adalah karakter khusus yang diawali dengan tanda *backslash* (\). Karakter ini digunakan untuk memasukkan instruksi yang tidak bisa diketik langsung sebagai teks biasa.

Berikut adalah daftar yang paling sering digunakan:

Karakter	Nama	Kegunaan
\n	Newline	Berpindah ke baris baru.
\t	Horizontal Tab	Memberikan jarak horizontal (seperti menekan tombol Tab).
\"	Double Quote	Menampilkan tanda kutip dua di dalam string.
\\	Backslash	Menampilkan satu tanda <i>backslash</i> .

Contoh Penggunaan:

```
1 Console.WriteLine("Data Mahasiswa:\n\t\"Budi\"\n\t\"Susi\"");
```

2. Verbatim Strings (@)

Jika Anda memiliki teks yang mengandung banyak *backslash* (seperti alamat folder di Windows), menggunakan \\ berulang kali akan melelahkan. Anda bisa menggunakan simbol @ di depan kutip dua. Ini disebut **Verbatim String**, di mana C# akan mencetak teks **apa adanya**.

Contoh:

```
1 // Tanpa @
2 string path1 = "C:\\Users\\Dokumen\\Project";
3
4 // Dengan @ (Lebih rapi)
5 string path2 = @"C:\Users\Dokumen\Project";
```

3. Formatting Output (Angka & Mata uang)

Seringkali kita perlu menampilkan angka dengan format tertentu, misalnya sebagai mata uang atau membatasi jumlah desimal. Kita bisa menyisipkan kode format di dalam kurung kurawal { } saat menggunakan *String Interpolation*.

Kode Format	Nama	Hasil Contoh
:C	Currency	Menampilkan format mata uang (Rp, \$, dll).
:N3	Number	Menampilkan angka dengan ribuan dan 3 desimal.
:P	Percent	Mengubah angka menjadi persentase (0.5 -> 50%).

Contoh Kode:

```
1 decimal harga = 150000.50m;
2 double diskon = 0.15;
3
4 Console.WriteLine($"Harga Barang: {harga:C}");
5 Console.WriteLine($"Diskon: {diskon:P}");
6 Console.WriteLine($"Total setelah diskon: {harga * (decimal)(1 - diskon):N2}");
```

4. Perataan Teks (Alignment)

Anda juga bisa mengatur lebar teks agar sejajar (seperti tabel) dengan memberikan koma , diikuti angka lebar kolom.

- **Angka positif:** Rata kanan.
- **Angka negatif:** Rata kiri.

Contoh:

```
1 Console.WriteLine($"{"Barang",-10} | {"Harga",10}");
2 Console.WriteLine($"{"Kopi",-10} | {5000,10:C}");
3 Console.WriteLine($"{"Roti",-10} | {12000,10:C}");
```

- **3.3 Menulis Komentar: Single-line, Multi-line, dan XML Documentation**

- 1. Single-line Comment (//)**

Ini adalah jenis komentar yang paling umum digunakan untuk penjelasan singkat pada satu baris kode. Segala sesuatu di sebelah kanan tanda // akan dianggap sebagai komentar.

Contoh:

```
1 int skor = 100; // Inisialisasi skor awal pemain
2 // Console.WriteLine("Baris ini tidak akan dijalankan");
3 skor += 10;
```

- 2. Multi-line Comment (/* ... */)**

Jika Anda perlu menulis penjelasan yang panjang atau ingin menonaktifkan blok kode yang besar sementara waktu, gunakan komentar multi-baris.

Komentar ini dimulai dengan /* dan diakhiri dengan */.

Contoh:

```
1 /* Program ini dibuat oleh Budi
2    Versi: 1.0
3    Tujuan: Menghitung total pajak tahunan
4    berdasarkan pendapatan bruto.
5    */
6 int pendapatan = 5000000;
```

- 3.XML Documentation Comment (///)**

Ini adalah fitur spesial di C#. Komentar ini digunakan di atas sebuah **Class** atau **Method** untuk memberikan dokumentasi teknis. Kelebihannya, saat Anda mengarahkan kursor (*hover*) ke nama method tersebut di bagian lain kode, Visual Studio akan menampilkan penjelasan yang Anda tulis di sini. Cukup ketik /// (tiga kali garis miring) tepat di atas method, dan editor akan otomatis membuatkan struktur XML-nya.

Contoh:

```
1 /// <summary>
2 /// Menghitung luas persegi panjang.
3 /// </summary>
4 /// <param name="panjang">Nilai panjang dalam cm.</param>
5 /// <param name="lebar">Nilai lebar dalam cm.</param>
6 /// <returns>Hasil perkalian panjang dan lebar.</returns>
7 int HitungLuas(int panjang, int lebar)
8 {
9     return panjang * lebar;
10 }
```

Ringkasan Perbedaan

Jenis Komentar	Simbol	Kegunaan Utama
Single-line	//	Penjelasan singkat satu baris atau mematikan satu baris kode.
Multi-line	/* ... */	Penjelasan panjang atau mematikan satu blok kode besar.
XML Doc	///	Membuat dokumentasi resmi yang muncul di fitur <i>IntelliSense</i> .

Tips "Clean Code" dalam Menulis Komentar:

1. **Jangan menjelaskan hal yang sudah jelas:**
 - o Buruk: `int x = 5; // Membuat variabel x bernilai 5`
 - o Baik: `int x = 5; // Menyimpan batas percobaan login maksimal`
2. **Gunakan komentar untuk menjelaskan "MENGAPA":** Kode memberitahu komputer "bagaimana" cara melakukannya, komentar memberitahu manusia "mengapa" Anda melakukannya dengan cara tersebut.
3. **Hapus kode sampah:** Jangan biarkan kode yang tidak terpakai menumpuk di dalam komentar multi-baris. Jika sudah tidak dipakai, hapus saja (Anda selalu punya *Version Control* seperti Git).

Bab 4: Variabel dan Konstanta

• 4.1 Deklarasi dan Inisialisasi Variabel

1. Apa itu Deklarasi?

Deklarasi adalah proses memesan tempat di memori dengan menentukan **Tipe Data** dan **Nama Variabel**. Pada tahap ini, variabel sudah ada di sistem tetapi belum memiliki nilai yang bermakna (kosong).

Sintaks:

TipeData namaVariabel;

Contoh:

```

1  int jumlahBarang;
2  string namaLengkap;
3  bool isLulus;

```

2. Apa itu Inisialisasi?

Inisialisasi adalah proses memberikan nilai awal ke dalam variabel yang sudah dideklarasikan menggunakan operator penugasan (=).

Contoh:

```

1  jumlahBarang = 50;
2  namaLengkap = "Budi Santoso";
3  isLulus = true;

```

3. Deklarasi dan Inisialisasi dalam Satu Baris

Untuk efisiensi, Anda sangat disarankan untuk melakukan keduanya secara bersamaan. Ini adalah cara yang paling sering digunakan oleh para pengembang karena kodenya lebih ringkas.

Sintaks:

TipeData namaVariabel = nilaiBaru;

Contoh:

```

1  int umur = 20;
2  double hargaKopi = 15000.50;
3  char inisial = 'B';

```

4. Penggunaan Kata Kunci var (Implicit Typing)

C# memiliki fitur di mana Anda tidak perlu menuliskan tipe datanya secara eksplisit. Komputer akan "menebak" tipe datanya berdasarkan nilai yang Anda berikan. Namun, Anda **wajib** langsung memberikan nilai (inisialisasi) saat itu juga agar komputer tidak bingung.

Contoh:

```

1  var pesan = "Selamat Pagi"; // Komputer tahu ini adalah string
2  var angka = 100;           // Komputer tahu ini adalah int
3  var diskon = 0.5;          // Komputer tahu ini adalah double

```

5. Aturan Penting Variabel

-Harus Diberi Nilai: C# tidak akan membiarkan Anda menggunakan variabel yang belum diberi nilai (*Unassigned variable*). Program akan protes (*error*) sebelum sempat dijalankan.

-Case-Sensitive: namaSiswa dan Namasiswa dianggap sebagai dua variabel yang berbeda.

-Karakter Pertama: Nama variabel tidak boleh diawali dengan angka.

-Keyword: Anda tidak boleh menggunakan kata kunci bawaan C# (seperti class, static, void) sebagai nama variabel Anda sendiri.

6. Contoh Program Gabungan

```
1 // Deklarasi & Inisialisasi
2 string namaHero = "Alucard";
3 int level = 15;
4 double damage = 1250.45;
5 bool isDead = false;
6
7 // Mengubah nilai variabel (Re-assignment)
8 level = 16;
9
10 // Menampilkan ke layar menggunakan String Interpolation
11 Console.WriteLine($"Hero: {namaHero}");
12 Console.WriteLine($"Level: {level}");
13 Console.WriteLine($"Damage: {damage}");
14 Console.WriteLine($"Status Mati: {isDead}");
```

● 4.2 Aturan Penamaan Identifier

Dalam pemrograman (termasuk bahasa seperti C#, Java, Python, dll), **identifier** adalah nama yang digunakan untuk memberi identitas pada variabel, fungsi, class, dan lainnya.

Berikut adalah **aturan penamaan identifier** yang umum digunakan:

1. Harus diawali huruf atau underscore (_)

- Boleh: nama, _data, nilaiAkhir
- Tidak boleh: 1nilai, @data

2. Tidak boleh mengandung spasi

- Boleh: totalNilai, data_siswa
- Tidak boleh: total nilai

3. Tidak boleh menggunakan kata kunci (keyword)

Kata yang sudah digunakan oleh bahasa pemrograman tidak bisa dipakai.
Contoh:

- Tidak boleh: int, class, public

4. Bersifat case-sensitive (membedakan huruf besar dan kecil)

- nama, Nama, dan NAMA dianggap berbeda

5. Boleh mengandung angka, tapi tidak di awal

- Boleh: nilai1, data2024
- Tidak boleh: 123data

6. Tidak boleh menggunakan simbol khusus

- Tidak boleh: nilai@, data#, total!
- Kecuali underscore _

Tips Penamaan yang Baik

- Gunakan nama yang jelas dan mudah dipahami
Contoh: jumlahSiswa lebih baik daripada js
- Gunakan format:
 - **camelCase** → nilaiAkhir
 - **PascalCase** → NamaSiswa
 - **snake_case** → data_siswa

- **4.3 Konstanta dengan `const` dan `readonly`**

Dalam bahasa pemrograman seperti **C#**, ada dua cara untuk membuat nilai yang tidak berubah, yaitu **`const`** dan **`readonly`**. Keduanya mirip, tapi punya perbedaan penting.

- ◆ **1. `const` (Konstanta)**

`const` digunakan untuk nilai yang **tetap dan tidak bisa diubah sama sekali** sejak awal.

Ciri-ciri:

- Harus langsung diisi saat deklarasi
- Nilainya **tidak bisa diubah** selamanya
- Bersifat **compile-time constant** (ditentukan saat program dikompilasi)

Contoh:

```
const double PI = 3.14;
```

```
const int MAX_SISWA = 30;
```

Nilai PI dan MAX_SISWA tidak bisa diubah di bagian program manapun.

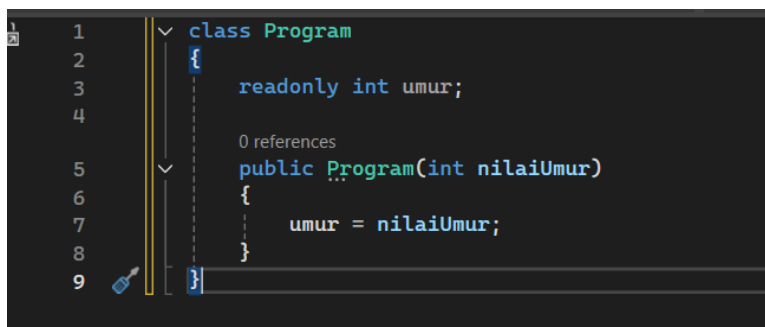
- ◆ **2. `readonly`**

`readonly` digunakan untuk nilai yang hanya bisa diisi **sekali**, tapi tidak harus langsung saat deklarasi.

Ciri-ciri:

- Bisa diisi saat deklarasi **atau di dalam constructor**
- Nilainya tidak bisa diubah setelah itu
- Bersifat **runtime constant**

Contoh:



```
1 class Program
2 {
3     readonly int umur;
4
5     0 references
6     public Program(int nilaiUmur)
7     {
8         umur = nilaiUmur;
9     }
10 }
```

Perbedaan `const` vs `readonly`

Aspek	<code>const</code>	<code>readonly</code>
Waktu penentuan	Compile-time	Runtime
Bisa diubah	Tidak sama sekali	Tidak (setelah inisialisasi)
Tempat isi nilai	Harus saat deklarasi	Bisa di constructor
Fleksibilitas	Kurang fleksibel	Lebih fleksibel

- **4.4 Multiple Variable Declaration**

Multiple Variable Declaration adalah cara mendeklarasikan **beberapa variabel sekaligus dalam satu baris** untuk tipe data yang sama, terutama dalam bahasa seperti **C#**.

♦ 1. Deklarasi tanpa nilai (inisialisasi)

Kita bisa membuat beberapa variabel sekaligus tanpa langsung memberi nilai:

```
int a, b, c;
```

Artinya kita membuat 3 variabel bertipe int: a, b, dan c

♦ 2. Deklarasi dengan nilai

Kita juga bisa langsung memberi nilai pada masing-masing variabel:

```
int a = 10, b = 20, c = 30;
```

Setiap variabel bisa punya nilai berbeda

♦ 3. Kombinasi (sebagian diberi nilai)

```
int a = 5, b, c = 15;
```

a dan c punya nilai, sedangkan b belum

♦ 4. Tipe data harus sama

Semua variabel dalam satu deklarasi **harus bertipe sama**:

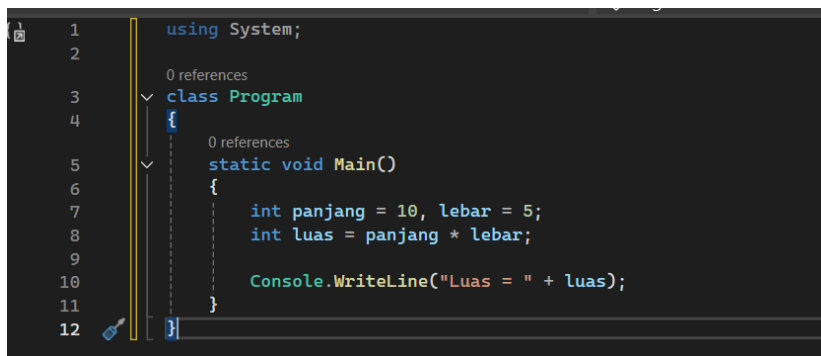
```
int x = 1, y = 2; // ✓ benar
```

```
string nama = "Ani", kota = "Bandung"; // ✓ benar
```

✗ Tidak boleh:

```
int x = 1, string nama = "Ani"; // salah
```

♦ 5. Contoh dalam program



```
1 using System;
2
3 class Program
4 {
5     static void Main()
6     {
7         int panjang = 10, lebar = 5;
8         int luas = panjang * lebar;
9
10        Console.WriteLine("Luas = " + luas);
11    }
12 }
```

♦ Kelebihan

- Lebih **ringkas dan cepat**
- Kode jadi lebih **rapi**

♦ Kekurangan

- Kalau terlalu banyak dalam satu baris, bisa jadi **sulit dibaca**

• 4.5 Scope Variabel: Local, Class, dan Global

♦ 1. Local Variable (Variabel Lokal)

Variabel yang dibuat **di dalam method/fungsi**.

Ciri-ciri:

- Hanya bisa digunakan di dalam method tersebut

- Tidak bisa diakses dari luar
- Umurnya hanya selama method berjalan

Contoh:

```

1  class Program
2  {
3      0 references
4      static void Main()
5      {
6          int angka = 10; // variabel lokal
7          Console.WriteLine(angka);
8      }

```

♦ 2. Class Variable (Field)

Variabel yang dibuat **di dalam class**, tapi di luar method.

Ciri-ciri:

- Bisa diakses oleh semua method dalam class
- Bisa dibuat private, public, dll
- Umurnya selama objek masih ada

Contoh:

```

1  class Program
2  {
3      int nilai = 100; // variabel class (field)
4
5      0 references
6      void Tampil()
7      {
8          Console.WriteLine(nilai);
9      }

```

♦ 3. Global Variable

Dalam C#, **tidak ada global variable** yang benar-benar bebas seperti di beberapa bahasa lain.

Tapi biasanya yang dimaksud:

- Variabel **class** yang dibuat **public static**
- Bisa diakses dari berbagai class

Contoh:

```

1  class GlobalData
2  {
3      public static int jumlah = 50;
4  }
5
6  0 references
7  class Program
8  {
9      0 references
10     static void Main()
11     {
12         Console.WriteLine(GlobalData.jumlah);
    }
}

```

♦ Perbandingan

Jenis	Letak	Bisa diakses di mana?
Local	Dalam method	Hanya di method itu
Class	Dalam class	Semua method dalam class
Global	Static di class lain	Bisa diakses antar class

♦ Kesimpulan

- **Local** → hanya untuk kebutuhan sementara dalam method
- **Class** → untuk data yang dipakai dalam satu class
- **Global (static)** → untuk data yang perlu diakses lebih luas

Bab 5: Tipe Data dalam C#

- 5.1 Tipe Data Primitif: `int`, `double`, `char`, `bool`, dll.

Dalam pemrograman, **tipe data primitif** adalah tipe data dasar yang digunakan untuk menyimpan nilai sederhana. Di bahasa seperti **C#**, tipe ini sering dipakai untuk membuat variabel.

♦ Jenis Tipe Data Primitif

1. int (Integer)

Digunakan untuk menyimpan **bilangan bulat**.

Contoh:

```
int umur = 16;
```

int jumlahSiswa = 30;

2. double

Digunakan untuk menyimpan **bilangan desimal** (pecahan).

Contoh:

double tinggi = 165.5;

double berat = 50.75;

3. char (Character)

Digunakan untuk menyimpan **satu karakter**.

Contoh:

char huruf = 'A';

char simbol = '#';

4. bool (Boolean)

Digunakan untuk menyimpan nilai **benar atau salah**.

Contoh:

bool isLulus = true;

bool isAktif = false;

5. string

Digunakan untuk menyimpan **teks atau kumpulan karakter**.

Contoh:

string nama = "Nabila";

string sekolah = "SMK";

6. Tipe lain (tambahan)

- float → desimal (lebih kecil dari double)
- long → bilangan bulat besar
- byte → bilangan kecil (0–255)

Contoh:

float nilai = 90.5f;

long populasi = 1000000;

byte umurAnak = 12;

♦ Perbandingan Singkat

Tipe	Kegunaan	Contoh Nilai
int	Bilangan bulat	10, 25
double	Bilangan desimal	3.14, 2.5
char	Satu karakter	'A', 'B'
bool	True / False	true
string	Teks	"Halo"

♦ Contoh Program

```
1  using System;
2
3  0 references
4  class Program
5  {
6      0 references
7      static void Main()
8      {
9          int umur = 16;
10         double tinggi = 165.5;
11         char nilai = 'A';
12         bool lulus = true;
13         string nama = "Nabila";
14
15         Console.WriteLine(nama);
16     }
17 }
```

♦ Kesimpulan

- Tipe data primitif digunakan untuk menyimpan data dasar
- Pemilihan tipe data harus sesuai dengan jenis nilai
- Salah memilih tipe bisa menyebabkan error atau hasil tidak akurat

• 5.2 Tipe Data Reference: **string**, **object**, **dynamic**

Dalam **C#**, selain tipe data primitif (value type), ada juga **tipe data reference**. Artinya, variabel tidak langsung menyimpan nilai, tetapi **menyimpan referensi (alamat) ke data di memori**.

Beberapa tipe reference yang sering digunakan: **string**, **object**, dan **dynamic**.

♦ 1. string

Digunakan untuk menyimpan **teks**.

Ciri-ciri:

- Termasuk reference type
- Bersifat **immutable** (tidak bisa diubah langsung)
- Jika diubah, sebenarnya dibuat objek baru

Contoh:

```
string nama = "Nabila";  
nama = "Salsa"; // membuat objek baru, bukan mengubah yang lama
```

♦ **2. object**

Merupakan **tipe dasar (induk)** dari **semua tipe data** di C#.

Ciri-ciri:

- Bisa menyimpan **semua jenis data**
- Perlu casting saat mengambil nilai spesifik

Contoh:

```
object data;
```

```
data = 10;    // int  
data = "Halo"; // string  
data = true;  // bool
```

```
int angka = (int)data; // perlu casting (jika tipe sesuai)
```

♦ **3. dynamic**

Digunakan untuk variabel yang **tipe datanya ditentukan saat runtime**.

Ciri-ciri:

- Tidak dicek saat compile
- Lebih fleksibel, tapi lebih berisiko error

Contoh:

```
1  dynamic data = 10;  
2  Console.WriteLine(data);  
3  
4  data = "Halo";  
5  Console.WriteLine(data);
```

♦ **Perbandingan**

Tipe	Fleksibilitas	Waktu Pengecekan	Catatan
string	Rendah	Compile-time	Khusus teks
object	Tinggi	Compile-time	Perlu casting
dynamic	Sangat tinggi	Runtime	Lebih rawan error

♦ **Perbedaan Utama**

- string → khusus teks
- object → semua tipe, tapi perlu casting
- dynamic → bebas tanpa casting, tapi dicek saat runtime

• **5.3 Nullable Types (`int?`, `bool?`)**

Dalam C#, Nullable Types memungkinkan tipe data value (seperti int, bool, dll) untuk menyimpan nilai null (kosong/tidak ada nilai).

Biasanya, tipe data seperti int atau bool tidak bisa null. Nah, dengan nullable, hal itu jadi bisa.

◆ 1. Penulisan Nullable

Nullable ditulis dengan tanda ? setelah tipe data.

```
1 int? umur = null;
2 bool? isAktif = null;
```

◆ 2. Kenapa Nullable Digunakan?

Kadang kita butuh kondisi:

- Nilai belum diisi
- Data belum diketahui
- Hasil dari database bisa kosong

Contoh:

```
1 int? nilaiUjian = null; // belum ikut ujian
```

◆ 3. Mengecek Nilai Nullable

a. Menggunakan .HasValue

```
1 if (umur.HasValue)
2 {
3     Console.WriteLine(umur.Value);
4 }
5 else
6 {
7     Console.WriteLine("Belum diisi");
8 }
```

```
1 if (umur == null)
2 {
3     Console.WriteLine("Kosong");
4 }
```

◆ 4. Memberi Nilai Default (??)

Operator ?? digunakan untuk memberi nilai cadangan.

```
1 int? umur = null;
2 int hasil = umur ?? 0;
3
4 Console.WriteLine(hasil); // output: 0
```

◆ 5. Nullable dengan Boolean

```

1  bool? status = null;
2
3  if (status == true)
4  {
5      Console.WriteLine("Aktif");
6  }
7  else if (status == false)
8  {
9      Console.WriteLine("Tidak Aktif");
10 }
11 else
12 {
13     Console.WriteLine("Belum diketahui");
14 }

```

• 5.4 Var Keyword dan Type Inference

Dalam **C#**, kata kunci **var** digunakan untuk **type inference**, yaitu ketika **tipe data** ditentukan otomatis oleh compiler berdasarkan nilai yang diberikan.

♦ 1. Apa itu var?

var bukan berarti “tanpa tipe”, tapi **tipe tetap ada**—hanya saja ditentukan otomatis.

Contoh:

var angka = 10; // dianggap int

var nama = "Nabila"; // dianggap string

var tinggi = 165.5; // dianggap double

👉 Compiler langsung tahu tipe datanya dari nilai di sebelah kanan

♦ 2. Aturan Penggunaan var

✅ **Harus langsung diinisialisasi**

var x = 5; // benar

❌ Tidak boleh tanpa nilai

var x; // salah

✅ **Tipe tidak bisa berubah**

var data = 10;

data = 20; // boleh

// data = "Halo"; ❌ salah (beda tipe)

Hanya untuk variabel lokal

var hanya bisa digunakan di dalam method (bukan untuk field/class)

♦ 3. Type Inference (Penentuan Tipe Otomatis)

Type inference adalah proses saat compiler **menebak tipe data** dari nilai.

Contoh:

♦ 4. Kapan Sebaiknya Pakai var?

Cocok digunakan:

- Tipe sudah jelas dari nilai
var nama = "Salsa";

- Tipe panjang/rumit
var daftar = new List<string>();
- **Kurang cocok:**
- Kalau bikin kode jadi sulit dibaca
var x = GetData(); // kurang jelas tipe datanya

♦ Perbandingan

Penulisan	Keterangan
int x = 10;	Tipe ditulis jelas
var x = 10;	Tipe otomatis (int)

- **5.5 Memeriksa Tipe Data dengan `typeof()` dan `GetType()`**

Bab 6: Type Casting dan Konversi

- 6.1 Implicit vs Explicit Casting Dalam **C#**, kita bisa **memeriksa tipe data** menggunakan dua cara utama: **`typeof()`** dan **`GetType()`**. Keduanya mirip, tapi digunakan dalam situasi yang berbeda.
- ♦ **1. `typeof()`**
- Digunakan untuk **mengetahui tipe data dari sebuah tipe (bukan variabel langsung)**.
- **Ciri-ciri:**
 - Ditulis dengan nama tipe
 - Diketahui saat **compile-time**
- **Contoh:**

```
1 Console.WriteLine(typeof(int));
2 Console.WriteLine(typeof(string));
```

- ♦ **2. `GetType()`**
 - Digunakan untuk **mengetahui tipe data dari sebuah objek/variabel**.
- **Ciri-ciri:**
 - Dipanggil dari variabel
 - Diketahui saat **runtime**

- **Contoh:**

```

1  int angka = 10;
2  string nama = "Nabila";
3
4  Console.WriteLine(angka.GetType());
5  Console.WriteLine(nama.GetType());

```

- **Perbedaan typeof() vs GetType()**

Aspek	typeof()	GetType()
Digunakan untuk	Tipe data	Variabel/objek
Waktu	Compile-time	Runtime
Penulisan	typeof(int)	variabel.GetType()

- **3. Contoh Perbandingan**

```

1  int x = 5;
2
3  Console.WriteLine(typeof(int)); // tipe langsung
4  Console.WriteLine(x.GetType()); // dari variabel

```

- **4. Catatan Penting**

- typeof() tidak bisa dipakai langsung ke nilai:
- // typeof(10); ❌ salah
- GetType() hanya bisa dipakai jika variabel sudah punya nilai (tidak null)

- **6.2 Konversi dengan Convert, Parse, dan TryParse**

Dalam **C#**, konversi tipe data sering dilakukan menggunakan **Convert**, **Parse**, dan **TryParse**. Ketiganya sama-sama mengubah data (misalnya dari string ke int), tapi cara kerjanya berbeda.

- **1. Convert**

Digunakan untuk mengubah tipe data dengan **lebih fleksibel**, termasuk menangani null.

Contoh:

```
1  string angka = "10";
2  int hasil = Convert.ToInt32(angka);
3
4  Console.WriteLine(hasil);
```

Ciri-ciri:

- Bisa menangani null → hasil jadi 0 (untuk angka)
- Mendukung banyak tipe (ToInt32, ToDouble, dll)

♦ 2. Parse

Digunakan untuk mengubah **string** ke tipe tertentu, tapi harus valid.

Contoh:

```
1  string angka = "20";
2  int hasil = int.Parse(angka);
3
4  Console.WriteLine(hasil);
```

Ciri-ciri:

- Harus berupa string yang valid
- Jika salah format → **error (exception)**

✗ Contoh error:

```
int hasil = int.Parse("abc"); // error
```

♦ 3. TryParse

Cara paling aman untuk konversi, karena **tidak menyebabkan error**.

Contoh:

```
1  string angka = "30";
2  int hasil;
3
4  bool sukses = int.TryParse(angka, out hasil);
5
6  if (sukses)
7  {
8      Console.WriteLine(hasil);
9  }
10 else
11 {
12     Console.WriteLine("Gagal konversi");
13 }
```

Ciri-ciri:

- Tidak error walaupun gagal
- Mengembalikan true atau false
- Menggunakan out parameter

♦ Perbandingan

Metode	Jika data salah	Kelebihan
Convert	Tidak error (null aman)	Fleksibel

Metode	Jika data salah	Kelebihan
Parse	Error	Cepat & langsung
TryParse	Tidak error	Paling aman

♦ Kapan Digunakan?

- **Convert** → data mungkin null
- **Parse** → data sudah pasti benar
- **TryParse** → data belum pasti (input user)

• 6.3 Boxing dan Unboxing

Dalam **C#**, **boxing** dan **unboxing** adalah proses konversi antara **value type** (seperti int, bool) dan **reference type** (object).

♦ 1. Boxing

Boxing adalah proses mengubah **value type** → **object (reference type)**.

Contoh:

```
1 int angka = 10;
2 object obj = angka; // boxing
```

Ciri-ciri:

- Terjadi otomatis (implicit)
- Membungkus value type ke dalam object
- Disimpan di heap (memori reference)

♦ 2. Unboxing

Unboxing adalah proses mengubah kembali **object** → **value type**.

Contoh:

```
1 object obj = 10;
2 int angka = (int)obj; // unboxing
```

Ciri-ciri:

- Tidak otomatis (explicit)
- Harus sesuai tipe aslinya
- Jika salah tipe → error

♦ Contoh Lengkap

```

1  using System;
2
3  0 references
4  class Program
5  {
6      0 references
7      static void Main()
8      {
9          int nilai = 25;
10         // Boxing
11         object obj = nilai;
12         // Unboxing
13         int hasil = (int)obj;
14         Console.WriteLine(hasil);
15     }
16 }
17

```

♦ 3. Hal yang Perlu Diperhatikan Harus sesuai tipe saat unboxing

```

1  object obj = 10;
2  double d = (double)obj; // X error

```

♦ 4. Dampak Performa

- Boxing & unboxing bisa **memperlambat program**
- Karena melibatkan:
 - Alokasi memori baru
 - Proses konversi tambahan

♦ Perbandingan

Proses	Arah Konversi	Cara
Boxing	Value → Reference	Otomatis
Unboxing	Reference → Value	Casting

• 6.4 Best Practices dalam Konversi Tipe Data

Dalam C#, konversi tipe data itu sering dipakai—tapi kalau asal, bisa bikin error atau bahkan bug yang susah dilacak. Jadi ada beberapa best practices (praktik terbaik) yang perlu kamu pegang.

♦ 1. Gunakan TryParse untuk Input User

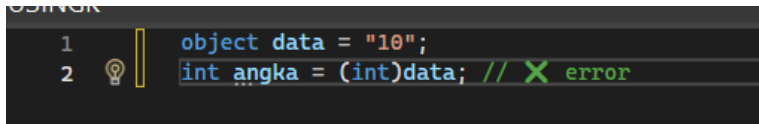
Kalau data berasal dari user (misalnya `Console.ReadLine()`), jangan langsung Parse.

```

1  int nilai;
2  if (int.TryParse(input, out nilai))
3  {
4      Console.WriteLine("Valid");
5  }
6  else
7  {
8      Console.WriteLine("Input salah");
9  }

```

♦ 2. Hindari Casting Sembarangan



```

1  object data = "10";
2  int angka = (int)data; // X error

```

✓ Lebih aman:

- ◆ 3. Perhatikan Nilai null

Gunakan Convert atau operator ?? untuk menangani null.

```
string data = null;
```

```
int angka = Convert.ToInt32(data); // hasil: 0
```

Atau:

```
int? nilai = null;
```

```
int hasil = nilai ?? 0;
```

- ◆ 4. Gunakan Tipe yang Sesuai Sejak Awal

Jangan sering konversi jika tidak perlu.

✗ Kurang baik:

```
string angka = "10";
```

```
int hasil = int.Parse(angka);
```

✓ Lebih baik:

```
int angka = 10;
```

- ◆ 5. Waspada Kehilangan Data (Data Loss)

Konversi dari tipe besar ke kecil bisa menghilangkan data.

```
double nilai = 9.8;
```

```
int hasil = (int)nilai; // jadi 9 (dibulatkan ke bawah)
```

- ◆ 6. Minimalkan Boxing & Unboxing

Terlalu sering boxing/unboxing bisa menurunkan performa.

✓ Gunakan tipe yang sesuai:

`int angka = 10; // lebih baik daripada object`

- ♦ 7. Gunakan var dengan Bijak

Kalau pakai var, pastikan tipe tetap jelas.

`var angka = 10; // jelas (int)`

✗ Kurang jelas:

`var data = GetValue(); // tipe tidak terlihat`

- ♦ 8. Validasi Sebelum Konversi

Selalu cek data sebelum dikonversi, terutama dari file atau database.

Bab 7: Input dari Pengguna

- 7.1 Membaca Input dengan `Console.ReadLine()`

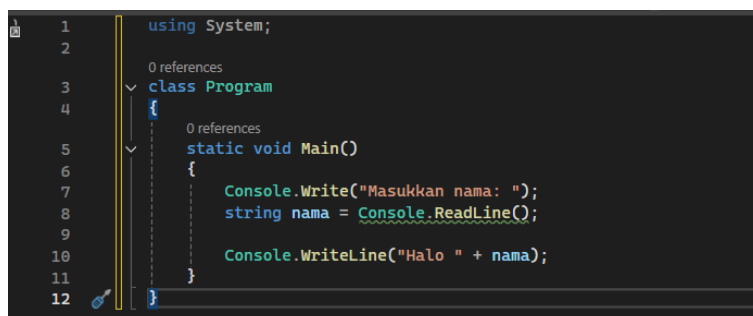
Dalam **C#**, `Console.ReadLine()` digunakan untuk **membaca input dari keyboard (user)** dalam bentuk **string**.

♦ 1. Dasar Penggunaan

`string input = Console.ReadLine();`

Program akan **menunggu user mengetik**, lalu menyimpan hasilnya ke variabel

♦ 2. Contoh Sederhana



```
1 using System;
2
3 class Program
4 {
5     static void Main()
6     {
7         Console.Write("Masukkan nama: ");
8         string nama = Console.ReadLine();
9
10        Console.WriteLine("Halo " + nama);
11    }
12 }
```

♦ 3. Input Selalu Bertipe String

Walaupun user memasukkan angka, hasilnya tetap **string**.

```
1 1 [ ] string angka = Console.ReadLine();
```

◆ 4. Mengubah Input ke Tipe Lain

a. Menggunakan Parse

```
1 1 [ ] int umur = int.Parse(Console.ReadLine());
```

b. Menggunakan TryParse (lebih aman)

```
1  int umur;
2  if (int.TryParse(Console.ReadLine(), out umur))
3  {
4      Console.WriteLine("Umur: " + umur);
5  }
6  else
7  {
8      Console.WriteLine("Input tidak valid");
9  }
```

◆ 5. Contoh Program Lengkap

```
1  using System;
2
3  0 references
4  class Program
5  {
6      0 references
7      static void Main()
8      {
9          Console.Write("Masukkan umur: ");
10         int umur;
11         if (int.TryParse(Console.ReadLine(), out umur))
12         {
13             Console.WriteLine("Umur kamu: " + umur);
14         }
15         else
16         {
17             Console.WriteLine("Harap masukkan angka!");
18         }
19     }
```

◆ 6. Tips Penting

- Console.ReadLine() selalu menghasilkan **string**
- Gunakan **TryParse** untuk menghindari error
- Pastikan user tahu apa yang harus diinput (beri petunjuk)

• 7.2 Konversi Input String ke Tipe Data Lain

Dalam **C#**, semua input dari `Console.ReadLine()` selalu berupa **string**, jadi kalau ingin digunakan sebagai angka atau tipe lain, harus **dikonversi** terlebih dahulu.

♦ 1. Menggunakan Parse

```
1 Console.Write("Masukkan umur: ");
2 int umur = int.Parse(Console.ReadLine());
3
4 Console.WriteLine("Umur: " + umur);
```

♦ 2. Menggunakan Convert

```
1 Console.Write("Masukkan tinggi: ");
2 double tinggi = Convert.ToDouble(Console.ReadLine());
3
4 Console.WriteLine("Tinggi: " + tinggi);
```

♦ 3. Menggunakan TryParse (Paling Aman)

```
1 Console.Write("Masukkan nilai: ");
2 int nilai;
3
4 if (int.TryParse(Console.ReadLine(), out nilai))
5 {
6     Console.WriteLine("Nilai: " + nilai);
7 }
8 else
9 {
10     Console.WriteLine("Input tidak valid!");
11 }
```

♦ 4. Contoh Konversi ke Berbagai Tipe

```
1 int angka = int.Parse("10");
2 double desimal = double.Parse("3.14");
3 bool status = bool.Parse("true");
4 char huruf = char.Parse("A");
```

♦ 5. Contoh Program Lengkap

```
1 using System;
2
3 class Program
4 {
5     static void Main()
6     {
7         Console.Write("Masukkan angka pertama: ");
8         int a = int.Parse(Console.ReadLine());
9
10        Console.Write("Masukkan angka kedua: ");
11        int b = int.Parse(Console.ReadLine());
12
13        int hasil = a + b;
14
15        Console.WriteLine("Hasil: " + hasil);
16    }
17 }
```

♦ 6. Tips Penting

- Gunakan **TryParse** untuk input dari user
- Hindari **Parse** jika data belum pasti
- Pastikan format input sesuai (angka, huruf, dll)

• 7.3 Validasi Input Dasar

Dalam **C#**, **validasi input dasar** adalah proses **mengecek apakah data yang dimasukkan user sudah benar sebelum diproses**. Ini penting supaya program tidak error dan hasilnya tetap akurat.

◆ 1. Cek Input Kosong

Pastikan user tidak mengirim input kosong.

```
1 string nama = Console.ReadLine();
2
3 if (string.IsNullOrEmpty(nama))
4 {
5     Console.WriteLine("Nama tidak boleh kosong!");
6 }
```

◆ 2. Validasi Angka dengan TryParse

```
1 int umur;
2 if (int.TryParse(Console.ReadLine(), out umur))
3 {
4     Console.WriteLine("Umur valid");
5 }
6 else
7 {
8     Console.WriteLine("Masukkan angka yang benar!");
9 }
```

◆ 3. Validasi Rentang Nilai

Pastikan nilai sesuai batas tertentu.

```
1 int nilai;
2 if (int.TryParse(Console.ReadLine(), out nilai) && nilai >= 0 && nilai <= 100)
3 {
4     Console.WriteLine("Nilai valid");
5 }
6 else
7 {
8     Console.WriteLine("Nilai harus 0 - 100");
9 }
```

◆ 4. Validasi Pilihan (Ya/Tidak, dll)

```
1 string jawaban = Console.ReadLine().ToLower();
2
3 if (jawaban == "ya" || jawaban == "tidak")
4 {
5     Console.WriteLine("Input valid");
6 }
7 else
8 {
9     Console.WriteLine("Masukkan 'ya' atau 'tidak'");
10 }
```

◆ 5. Mengulang Input Sampai Benar

```

1  int umur;
2  while (!int.TryParse(Console.ReadLine(), out umur))
3  {
4      Console.WriteLine("Input salah, coba lagi:");
5  }

```

♦ 6. Contoh Program Lengkap

```

1  using System;
2
3  0 references
4  class Program
5  {
6      0 references
7      static void Main()
8      {
9          int umur;
10         Console.Write("Masukkan umur: ");
11         while (!int.TryParse(Console.ReadLine(), out umur))
12         {
13             Console.Write("Input tidak valid, masukkan angka: ");
14         }
15         Console.WriteLine("Umur kamu: " + umur);
16     }
17 }

```

♦ Tips Penting

- Selalu validasi input dari user
- Gunakan TryParse untuk angka
- Gunakan kondisi (if) untuk aturan tertentu
- Gunakan loop agar input benar

Bab 8: Operator dalam C#

• 8.1 Operator Aritmatika (+, -, *, /, %, ++, --)

Operasi aritmatika adalah fondasi dari hampir semua logika pemrograman. Operator ini digunakan untuk melakukan perhitungan matematika standar pada data numerik (seperti integer atau float).

Berikut adalah rincian lengkap mengenai cara kerja masing-masing operator:

1. Operator Aritmatika Dasar

Operator ini bekerja pada dua nilai (operand) untuk menghasilkan nilai baru.

Operator	Nama	Penjelasan	Contoh (x=10,y=3)
+	Penjumlahan	Menambahkan dua nilai.	$x + y = 13$
-	Pengurangan	Mengurangi nilai kanan dari nilai kiri.	$x - y = 7$
*	Perkalian	Mengalikan dua nilai.	$x * y = 30$
/	Pembagian	Membagi nilai kiri dengan nilai kanan.	$x / y = 3.33$
%	Modulus	Menghasilkan sisa bagi dari pembagian.	$x \% y = 1$

2. Memahami Modulus (%)

Modulus sering kali membingungkan bagi pemula. Bayangkan Anda memiliki 10 apel dan ingin membaginya ke 3 orang secara merata.

- Setiap orang mendapat 3 apel (Total 9 apel digunakan).
- Sisa **1 apel** yang tidak bisa dibagi rata itulah yang disebut **Modulus**.

Tips: Modulus sangat berguna untuk mengecek apakah sebuah angka ganjil atau genap. Jika $\text{Angka} \% 2 == 0$, maka angka tersebut genap.

3. Operator Increment & Decrement

Operator ini digunakan untuk menambah atau mengurangi nilai variabel sebanyak **1**.

- **Increment (++)**: Menambah nilai sebesar 1.
 - Jika $x = 5$, maka setelah $x++$, nilai x menjadi **6**.
- **Decrement (--)**: Mengurangi nilai sebesar 1.
 - Jika $x = 5$, maka setelah $x--$, nilai x menjadi **4**.

Perbedaan Prefix dan Postfix:

Dalam banyak bahasa pemrograman (seperti C++, Java, atau JavaScript), posisi operator ini berpengaruh:

1. **Prefix (++)**: Tambahkan nilainya dulu, baru gunakan variabelnya dalam ekspresi.
2. **Postfix (x++)**: Gunakan nilai variabel yang sekarang dalam ekspresi, baru kemudian tambahkan nilainya.

Urutan Operasi (Precedence)

Sama seperti matematika di sekolah, pemrograman mengikuti aturan **PEMDAS/KABATAKU**. Perkalian dan pembagian akan selalu dijalankan lebih dahulu sebelum penjumlahan dan pengurangan, kecuali jika Anda menggunakan tanda kurung ().

$Hasil = 5 + 2 * 10 \rightarrow 25$

$Hasil = (5 + 2) * 10 \rightarrow 70$

• 8.2 Operator Assignment (=, +=, -=, dll.)

Operator **Assignment** (Penugasan) digunakan untuk memberikan nilai ke dalam sebuah variabel. Di dalam pemrograman, operator ini tidak hanya sekadar "sama dengan", tetapi juga bisa dikombinasikan dengan operator aritmatika untuk menyingkat penulisan kode (disebut *Shorthand Assignment*).

1. Operator Assignment Dasar (=)

Ini adalah operator yang paling sering digunakan. Ia mengambil nilai di sisi kanan dan menyimpannya ke variabel di sisi kiri.

- **Contoh:** $x = 10$ artinya nilai 10 dimasukkan ke dalam variabel x.

2. Compound Assignment (Shorthand)

Operator ini menggabungkan operasi aritmatika dengan penugasan dalam satu langkah. Ini membuat kode lebih bersih dan efisien.

Operator	Nama	Contoh	Padanan (Sama Dengan)
+=	Addition Assignment	<code>x += 5</code>	<code>x = x + 5</code>
-=	Subtraction Assignment	<code>x -= 2</code>	<code>x = x - 2</code>
*=	Multiplication Assignment	<code>x *= 3</code>	<code>x = x * 3</code>
/=	Division Assignment	<code>x /= 4</code>	<code>x = x / 4</code>
%=	Remainder Assignment	<code>x %= 3</code>	<code>x = x % 3</code>

3. Mengapa Menggunakan Shorthand?

Selain karena lebih cepat diketik, operator shorthand sering kali lebih optimal secara performa di beberapa bahasa pemrograman karena komputer hanya perlu mengevaluasi lokasi memori variabel satu kali saja.

Analogi:

- **Cara Panjang:** "Ambil isi kotak A, tambahkan 5, lalu masukkan kembali ke kotak A."
- **Cara Shorthand (+=):** "Tambahkan 5 ke dalam kotak A."

Contoh Implementasi dalam Kode

Bayangkan Anda sedang membuat sistem skor untuk permainan:

```

1  let score = 100;
2
3  // Pemain mendapat bonus 50 poin
4  score += 50; // score sekarang 150
5
6  // Pemain terkena penalti 20 poin
7  score -= 20; // score sekarang 130
8
9  // Skor digandakan karena power-up
10 score *= 2; // score sekarang 260

```

• 8.3 Operator Perbandingan (**==**, **!=**, **>**, **<**, **>=**, **<=**)

Operator **Perbandingan** (atau sering disebut operator relasional) digunakan untuk membandingkan dua nilai. Hasil dari perbandingan ini selalu berupa nilai **Boolean**, yaitu **true** (benar) atau **false** (salah).

Operator ini adalah kunci utama dalam pembuatan alur logika, seperti pada pernyataan if-else atau perulangan.

Daftar Operator Perbandingan

Operator	Nama	Penjelasan	Contoh (x=10,y=3)	Hasil
==	Sama Dengan	Memeriksa apakah dua nilai sama.	\$x == y\$	false
!=	Tidak Sama Dengan	Memeriksa apakah dua nilai berbeda.	\$x != y\$	true
>	Lebih Besar Dari	Memeriksa apakah nilai kiri lebih besar dari kanan.	\$x > y\$	true
<	Lebih Kecil Dari	Memeriksa apakah nilai kiri lebih kecil dari kanan.	\$x < y\$	false
>=	Lebih Besar atau Sama Dengan	Memeriksa apakah nilai kiri lebih besar atau sama.	\$x >= 10\$	true
<=	Lebih Kecil atau Sama Dengan	Memeriksa apakah nilai kiri lebih kecil atau sama.	\$x <= 5\$	false

Perbedaan Penting: = vs ==

Salah satu kesalahan paling umum bagi pemula adalah tertukar antara satu tanda sama dengan dan dua tanda sama dengan:

- **= (Assignment):** Digunakan untuk **mengisi** nilai ke variabel. (Contoh: x = 5)
- **== (Comparison):** Digunakan untuk **mempertanyakan** apakah nilainya sama. (Contoh: x == 5?)

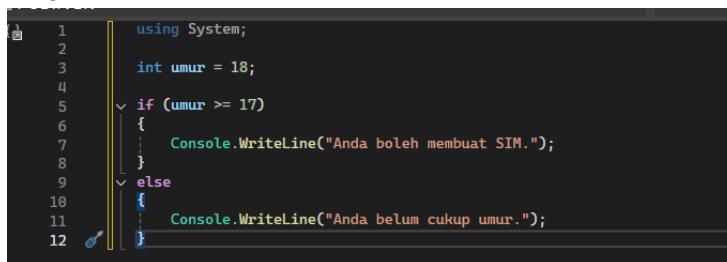
Catatan Khusus: Strict Equality (===)

Dalam beberapa bahasa pemrograman modern seperti **JavaScript** atau **PHP**, terdapat operator tambahan yaitu **===** (Identik).

- **==**: Hanya membandingkan **nilainya** saja. (Contoh: 5 == "5" adalah true).
 - **===**: Membandingkan **nilai DAN tipe datanya**. (Contoh: 5 === "5" adalah false karena yang satu angka, yang satu teks/string).
-

Contoh Penggunaan dalam Logika

Operator perbandingan biasanya menjadi "pertanyaan" yang menentukan arah program:



```
1 using System;
2
3 int umur = 18;
4
5 if (umur >= 17)
6 {
7     Console.WriteLine("Anda boleh membuat SIM.");
8 }
9 else
10 {
11     Console.WriteLine("Anda belum cukup umur.");
12 }
```

- **8.4 Operator Logika (&&, ||, !)**

Operator **Logika** digunakan untuk menghubungkan dua atau lebih ekspresi perbandingan atau untuk membalikkan nilai logika. Operator ini sangat krusial dalam pembuatan keputusan yang kompleks, di mana satu syarat saja tidak cukup. Hasil akhirnya selalu berupa **Boolean** (true atau false).

1. Operator AND (&&)

Operator ini hanya akan menghasilkan **true** jika **semua** kondisi yang diperiksa bernilai benar. Jika ada satu saja yang salah, maka hasilnya false.

- **Analogi**: "Anda bisa masuk jika membawa **tiket** AND memakai **sepatu**." (Keduanya wajib ada).

Kondisi A	Kondisi B	Hasil (A && B)
true	true	true
true	false	false
false	true	false
false	false	false

2. Operator OR (||)

Operator ini akan menghasilkan **true** jika **salah satu** atau kedua kondisi bernilai benar. Ia hanya akan menghasilkan false jika semua kondisi salah.

- **Analogi:** "Anda bisa masuk jika membawa **paspor OR KTP**." (Salah satu saja sudah cukup).

Kondisi A	Kondisi B	Hasil (A B)
true	true	true
true	false	true
false	true	true
false	false	false

3. Operator NOT (!)

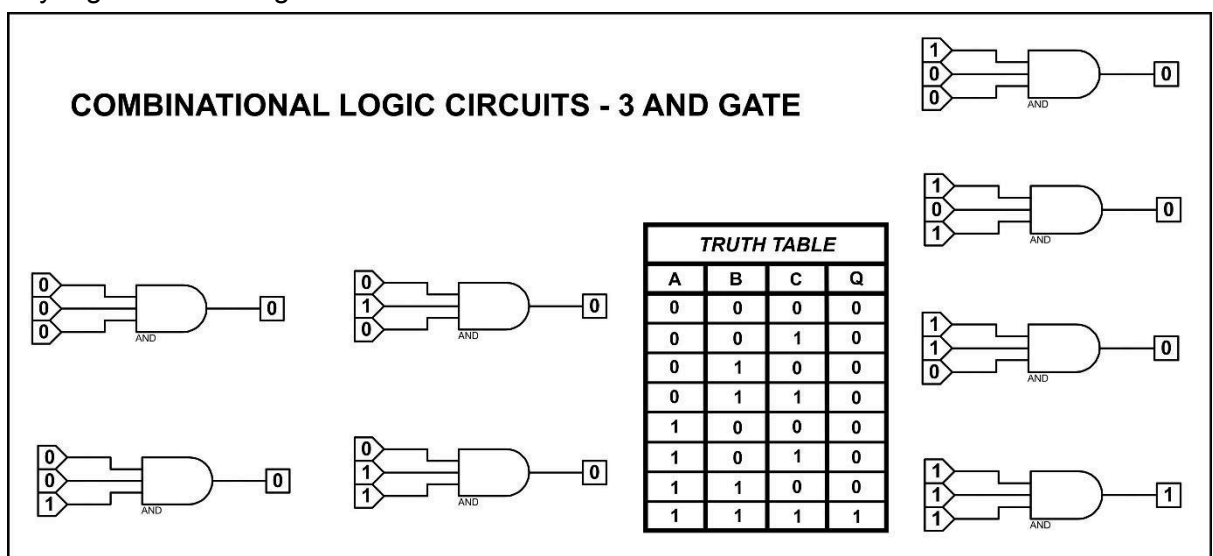
Operator ini digunakan untuk **membalikkan** nilai logika. Jika nilai awalnya true, maka menjadi false, dan sebaliknya.

- **Analogi:** "Bukan benar adalah salah."

Kondisi A	Hasil (!A)
true	false
false	true

Contoh Implementasi Gabungan

Bayangkan sistem login sebuah website:



Shutterstock

```

1  bool punyaKunci = true;
2  bool tahuPassword = false;
3
4  // AND (&&): Keduanya harus benar
5  if (punyaKunci && tahuPassword)
6  {
7      Console.WriteLine("Pintu Terbuka");
8  }
9  else
10 {
11     Console.WriteLine("Akses Ditolak"); // Ini yang muncul
12 }
13
14 // OR (||): Salah satu benar sudah cukup
15 if (punyaKunci || tahuPassword)
16 {
17     Console.WriteLine("Minimal ada satu akses"); // Ini yang muncul
18 }
19
20 // NOT (!): Membalikkan nilai
21 bool sedangHujan = true;
22 Console.WriteLine(!sedangHujan); // false

```

Tips Penting: Short-Circuit Evaluation

Komputer sangat efisien dalam memproses operator logika:

- Pada **&&**, jika kondisi pertama sudah false, komputer tidak akan mengecek kondisi kedua karena hasilnya pasti false.
- Pada **||**, jika kondisi pertama sudah true, komputer tidak akan mengecek kondisi kedua karena hasilnya pasti true.

• 8.5 Operator Bitwise dan Ternary (?:)

Ini adalah dua operator yang sangat berguna di C# untuk mengoptimalkan kode.

Bitwise bekerja di level biner (bit), sedangkan **Ternary** adalah cara singkat untuk menulis percabangan if-else.

1. Operator Ternary (?:)

Operator ini disebut "ternary" karena melibatkan tiga bagian. Ini adalah jalan pintas untuk memberikan nilai berdasarkan suatu kondisi.

Sintaks:

variabel = (kondisi) ? nilai_jika_true : nilai_jika_false;

Contoh dalam C#:

```

1  int nilai = 85;
2  string hasil = (nilai >= 75) ? "Lulus" : "Tidak Lulus";
3
4  Console.WriteLine(hasil); // Output: Lulus

```

Mengapa pakai ini? Kode jadi jauh lebih ringkas (1 baris) dibandingkan menggunakan if-else standar yang memakan 4-5 baris.

2. Operator Bitwise

Operator ini melakukan operasi pada representasi **biner** (0 dan 1) dari sebuah angka. Meskipun jarang digunakan dalam aplikasi bisnis biasa, ini sangat penting dalam pemrograman sistem, grafis, atau enkripsi.

Misalkan kita punya angka **5** (biner: 0101) dan **3** (biner: 0011):

Operator	Nama	Cara Kerja	Contoh (5 & 3)	Hasil
&	AND	1 jika kedua bit adalah 1	0101 & 0011	0001 (1)
	OR	1 jika salah satu bit adalah 1	0101 0011	0111 (7)
^	XOR	1 jika bit berbeda	0101 ^ 0011	0110 (6)
~	NOT	Membalikkan semua bit	~0101	1010
<<	Left Shift	Geser bit ke kiri (kali 2)	5 << 1	1010 (10)
>>	Right Shift	Geser bit ke kanan (bagi 2)	5 >> 1	0010 (2)

Perbedaan Penting: && vs &

Di C#, jangan sampai tertukar:

- **&& (Logical AND):** Digunakan untuk logika true/false (Boolean).
- **& (Bitwise AND):** Digunakan untuk operasi angka biner atau evaluasi logika tanpa *short-circuit* (memeriksa kedua sisi apa pun yang terjadi).

Contoh Implementasi Gabungan di C#

Mari kita buat kode yang mengecek apakah sebuah angka itu **Genap** atau **Ganjil** menggunakan Bitwise dan Ternary:

```

1  int angka = 7;
2
3  // Menggunakan Bitwise &: Angka ganjil selalu punya bit terakhir 1
4  // Menggunakan Ternary: Menentukan string hasil
5  string jenis = (angka & 1) == 1 ? "Ganjil" : "Genap";
6
7  Console.WriteLine($"{angka} adalah bilangan {jenis}");
8  // Output: 7 adalah bilangan Ganjil

```

// Output: 7 adalah bilangan Ganjil

Apakah Anda ingin saya jelaskan lebih lanjut tentang bagaimana **Left Shift (<<)** dan **Right Shift (>>)** sering digunakan untuk optimasi matematika yang super cepat?

- **8.6 Operator Null-coalescing (??) dan Null-conditional (?.)**

- Dalam C#, operator ini sangat populer karena membantu mencegah error klasik yang paling sering menghantui programmer: `NullReferenceException` (error saat mencoba mengakses data yang isinya kosong atau `null`).
-

• 1. Operator Null-conditional (?.)

- Operator ini memungkinkan Anda mengakses anggota (properti atau metode) dari sebuah objek hanya jika objek tersebut **tidak null**. Jika objeknya `null`, ia akan langsung mengembalikan `null` daripada melempar error.

- **Sintaks:** `objek?.Properti`

- **Contoh Kasus:**

- Bayangkan Anda memiliki kelas `User` yang memiliki properti `Profile`.

- C#

- `User user = null;`

-

- `// Cara Lama (Berisiko Error)`

- `// string bio = user.Profile.Bio; // Jika user null, program CRASH!`

-

- `// Cara Aman dengan Null-conditional`

- `string bio = user?.Profile?.Bio;`

-

- `Console.WriteLine(bio); // Output: (kosong/null), tapi program tidak crash.`

• 2. Operator Null-coalescing (??)

- Operator ini digunakan untuk memberikan **nilai default** jika sebuah ekspresi bernilai `null`. Ini seperti berkata: "Gunakan nilai ini, tapi kalau kosong, pakai nilai cadangan ini."

- **Sintaks:** `variabel ?? nilai_default`

- **Contoh Kasus:**

- C#

- `string namaInput = null;`

- `string namaTampilan = namaInput ?? "Guest";`

-

- `Console.WriteLine(namaTampilan); // Output: Guest`

• 3. Gabungan Keduanya (?. dan ??)

- Biasanya, kedua operator ini digunakan bersamaan untuk membuat kode yang sangat ringkas namun aman.

- C#

- `User user = null;`

-

- `// Ambil bio, jika user null atau bio null, gunakan "No Bio Available"`

- `string bio = user?.Profile?.Bio ?? "No Bio Available";`

-

- `Console.WriteLine(bio); // Output: No Bio Available`

• 4. Operator Null-coalescing Assignment (??=)

- Diperkenalkan pada versi C# yang lebih baru, operator ini mengisi variabel dengan nilai hanya jika variabel tersebut saat ini bernilai `null`.

- C#
 - `List<string> daftarNama = null;`
 -
 - `// Jika daftarNama null, buat List baru`
 - `daftarNama ??= new List<string>();`
 -
 - `daftarNama.Add("Budi");`
 -
-
- Perbandingan Visual

Operator	Nama	Kegunaan Utama
?.	Null-conditional	Mencegah error saat mengakses properti dari objek yang mungkin null.
??	Null-coalescing	Memberikan nilai cadangan jika hasilnya null.
??=	Null-assignment	Mengisi variabel hanya jika variabel tersebut masih null.

- Dengan memahami operator ini, kode C# Anda akan jauh lebih bersih dan tidak mudah "pecah" saat berhadapan dengan data yang tidak lengkap! Apakah Anda ingin lanjut membahas tentang tipe data yang bisa bernilai null (**Nullable Types**) di C#?

Bab 9: Kelas Math dan Fungsi Matematika

- **9.1 Fungsi Dasar:** `Math.Abs()`, `Math.Round()`, `Math.Sqrt()`

Di dalam C#, kelas Math menyediakan berbagai fungsi statis untuk melakukan operasi matematika tingkat lanjut yang tidak bisa dilakukan oleh operator dasar.

Berikut adalah penjelasan fungsi-fungsi dasar yang sering digunakan:

1. Math.Abs() (Absolute)

Fungsi ini digunakan untuk mendapatkan **nilai mutlak** dari sebuah angka. Sederhananya, ia mengubah angka negatif menjadi positif, sedangkan angka positif tetap positif.

- **Kegunaan:** Menghitung selisih atau jarak yang tidak boleh bernilai negatif.
- **Contoh:**

```

1 Console.WriteLine(Math.Abs(-10)); // Hasil: 10
2 Console.WriteLine(Math.Abs(5.5)); // Hasil: 5.5

```

2. Math.Round() (Rounding)

Fungsi ini digunakan untuk **membulatkan** angka desimal ke bilangan bulat terdekat atau ke jumlah digit desimal tertentu.

- **Aturan Dasar:** Jika angka di belakang koma < 0.5 maka dibulatkan ke bawah. Jika > 0.5 dibulatkan ke atas.
- **Catatan C#:** Secara default, C# menggunakan *Banker's Rounding* (jika tepat 0.5, ia membulatkan ke angka genap terdekat).
- **Contoh:**

```

1 Console.WriteLine(Math.Round(10.4)); // Hasil: 10
2 Console.WriteLine(Math.Round(10.6)); // Hasil: 11
3 Console.WriteLine(Math.Round(10.5)); // Hasil: 10 (Banker's Rounding ke genap)
4
5 // Membulatkan ke 2 angka di belakang koma
6 Console.WriteLine(Math.Round(3.14159, 2)); // Hasil: 3.14

```

3. Math.Sqrt() (Square Root)

Fungsi ini digunakan untuk menghitung **akar kuadrat** dari sebuah angka. Hasilnya selalu bertipe data double.

- **Rumus Matematis:** $\sqrt{x}$
- **Contoh:**

```

1 Console.WriteLine(Math.Sqrt(25)); // Hasil: 5
2 Console.WriteLine(Math.Sqrt(2)); // Hasil: 1.4142135623731

```

Peringatan: Jika Anda memasukkan angka negatif ke Math.Sqrt(), hasilnya adalah NaN (*Not a Number*), karena akar kuadrat bilangan negatif tidak terdefinisi dalam bilangan riil.

Tabel Ringkasan

Fungsi	Input	Output (Contoh)	Deskripsi
Math.Abs(x)	-5	5	Menghilangkan tanda negatif.
Math.Round(x)	2.7	3	Pembulatan ke bilangan terdekat.
Math.Sqrt(x)	16	4	Mencari akar kuadrat.
Math.Max(x, y)	10, 20	20	Mengambil nilai terbesar.
Math.Min(x, y)	10, 20	10	Mengambil nilai terkecil.
Math.Pow(x, y)	2, 3	8	Perpangkatan (2^3).

Contoh Implementasi Gabungan

Misalkan kita ingin menghitung sisi miring segitiga siku-siku (Pythagoras $c = \sqrt{a^2 + b^2}$) dan membulatkan hasilnya:

C#

```
double a = 3;
```

```
double b = 4;
```

```
double c = Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2));
```

```
Console.WriteLine($"Sisi miring: {Math.Round(c)}"); // Hasil: 5
```

Apakah Anda ingin mempelajari fungsi Math lainnya seperti trigonometri (Sin, Cos, Tan) atau logaritma?

Di dalam C#, kelas Math menyediakan berbagai fungsi statis untuk melakukan operasi matematika tingkat lanjut yang tidak bisa dilakukan oleh operator dasar.

Berikut adalah penjelasan fungsi-fungsi dasar yang sering digunakan:

1. Math.Abs() (Absolute)

Fungsi ini digunakan untuk mendapatkan **nilai mutlak** dari sebuah angka. Sederhananya, ia mengubah angka negatif menjadi positif, sedangkan angka positif tetap positif.

- **Kegunaan:** Menghitung selisih atau jarak yang tidak boleh bernilai negatif.
- **Contoh:**

C#

```
Console.WriteLine(Math.Abs(-10)); // Hasil: 10
```

```
Console.WriteLine(Math.Abs(5.5)); // Hasil: 5.5
```

2. Math.Round() (Rounding)

Fungsi ini digunakan untuk **membulatkan** angka desimal ke bilangan bulat terdekat atau ke jumlah digit desimal tertentu.

- **Aturan Dasar:** Jika angka di belakang koma < 0.5 maka dibulatkan ke bawah. Jika > 0.5 dibulatkan ke atas.
- **Catatan C#:** Secara default, C# menggunakan *Banker's Rounding* (jika tepat 0.5, ia membulatkan ke angka genap terdekat).
- **Contoh:**

C#

```
Console.WriteLine(Math.Round(10.4)); // Hasil: 10
```

```
Console.WriteLine(Math.Round(10.6)); // Hasil: 11
```

```
Console.WriteLine(Math.Round(10.5)); // Hasil: 10 (Banker's Rounding ke genap)
```

// Membulatkan ke 2 angka di belakang koma

```
Console.WriteLine(Math.Round(3.14159, 2)); // Hasil: 3.14
```

3. Math.Sqrt() (Square Root)

Fungsi ini digunakan untuk menghitung **akar kuadrat** dari sebuah angka. Hasilnya selalu bertipe data double.

- **Rumus Matematis:** $\sqrt{x}$
- **Contoh:**

C#

```
Console.WriteLine(Math.Sqrt(25)); // Hasil: 5
```

```
Console.WriteLine(Math.Sqrt(2)); // Hasil: 1.4142135623731
```

Peringatan: Jika Anda memasukkan angka negatif ke Math.Sqrt(), hasilnya adalah NaN (*Not a Number*), karena akar kuadrat bilangan negatif tidak terdefinisi dalam bilangan riil.

Tabel Ringkasan

Fungsi	Input	Output (Contoh)	Deskripsi
Math.Abs(x)	-5	5	Menghilangkan tanda negatif.
Math.Round(x)	2.7	3	Pembulatan ke bilangan terdekat.
Math.Sqrt(x)	16	4	Mencari akar kuadrat.

Fungsi	Input	Output (Contoh)	Deskripsi
Math.Max(x, y)	10, 20	20	Mengambil nilai terbesar.
Math.Min(x, y)	10, 20	10	Mengambil nilai terkecil.
Math.Pow(x, y)	2, 3	8	Perpangkatan (2^3).

Contoh Implementasi Gabungan

Misalkan kita ingin menghitung sisi miring segitiga siku-siku (Pythagoras $c = \sqrt{a^2 + b^2}$) dan membulatkan hasilnya:

C#

```
double a = 3;
```

```
double b = 4;
```

```
double c = Math.Sqrt(Math.Pow(a, 2) + Math.Pow(b, 2));
```

```
Console.WriteLine($"Sisi miring: {Math.Round(c)}"); // Hasil: 5
```

Apakah Anda ingin mempelajari fungsi Math lainnya seperti trigonometri (Sin, Cos, Tan) atau logaritma?

- **9.2 Konstanta Matematika:** `Math.PI`, `Math.E`

Selain fungsi perhitungan, kelas Math di C# juga menyediakan konstanta matematika yang sangat akurat. Konstanta ini bersifat *read-only* (tidak bisa diubah) dan memiliki presisi tinggi (tipe data double).

1. Math.PI

Math.PI mewakili rasio keliling lingkaran terhadap diameternya. Nilainya kurang lebih **3.1415926535897931**.

- **Kegunaan:** Menghitung luas lingkaran, keliling lingkaran, volume bola, atau konversi sudut ke radian.

- **Rumus Luas Lingkaran:** $L = \pi \times r^2$

Contoh Kode C#:

```
C#
double radius = 7;
double luas = Math.PI * Math.Pow(radius, 2);

Console.WriteLine($"Luas lingkaran: {luas}");
// Output: Luas lingkaran: 153.93804002589985
```

2. Math.E

Math.E mewakili basis logaritma natural, yang dikenal sebagai **Bilangan Euler**. Nilainya kurang lebih **2.7182818284590451**.

- **Kegunaan:** Digunakan dalam perhitungan pertumbuhan eksponensial, peluruhan radioaktif, bunga majemuk dalam finansial, dan statistik.

Contoh Kode C#:

```
C#
// Menghitung e pangkat 2 (e^2)
double hasilE = Math.Pow(Math.E, 2);

Console.WriteLine($"e pangkat 2: {hasilE}");
// Output: e pangkat 2: 7.38905609893065
```

Mengapa Menggunakan Math.PI Daripada 3.14?

1. **Presisi:** Mengetik 3.14 secara manual akan menghilangkan banyak digit desimal, yang menyebabkan hasil perhitungan menjadi tidak akurat pada skala besar.
 2. **Keterbacaan:** Math.PI langsung memberitahu pembaca kode bahwa Anda sedang menggunakan konstanta matematika standar.
 3. **Standar:** Mengikuti praktik terbaik (*best practice*) dalam pemrograman C#.
-

Studi Kasus: Menghitung Keliling Lingkaran

Mari kita gabungkan konstanta Math.PI dengan fungsi Math.Round() agar hasilnya lebih enak dibaca:

```
C#
double d = 10; // diameter
double keliling = Math.PI * d;

// Bulatkan ke 2 angka di belakang koma
Console.WriteLine($"Keliling: {Math.Round(keliling, 2)} cm");
// Output: Keliling: 31.42 cm
```

Apakah Anda ingin melihat bagaimana konstanta ini digunakan dalam rumus yang lebih kompleks, seperti menghitung volume bola atau luas permukaan tabung?

- **9.3 Fungsi Trigonometri dan Eksponensial**

Dalam C#, kelas Math juga menyediakan fungsi lengkap untuk perhitungan geometri (trigonometri) dan pertumbuhan (eksponensial). Satu hal penting yang perlu diingat: C# menggunakan satuan **Radian**, bukan Derajat (Degree).

1. Fungsi Trigonometri

Fungsi-fungsi ini sangat berguna dalam pengembangan game (rotasi karakter), grafis komputer, dan teknik.

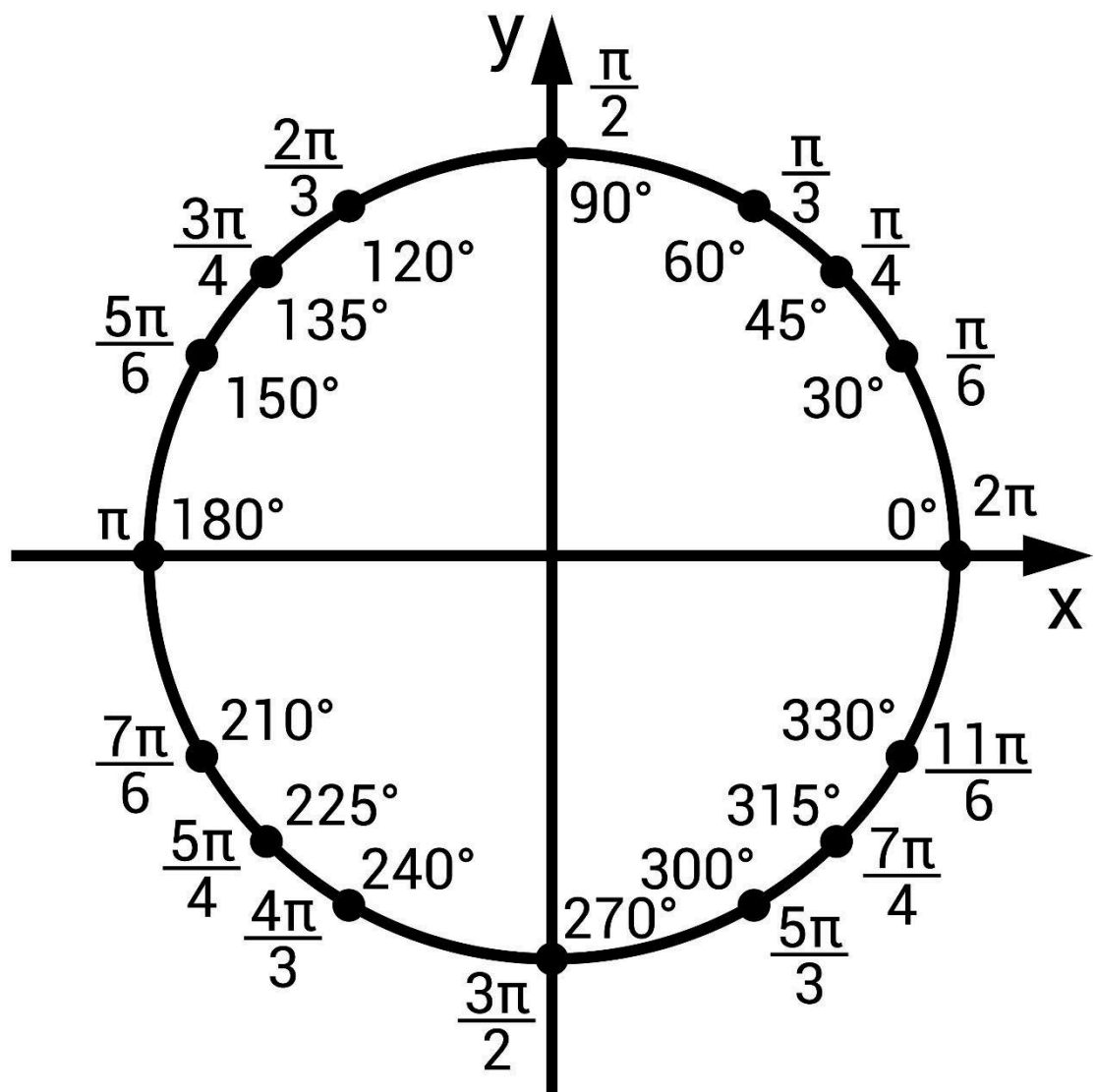
Fungsi	Deskripsi	Rumus Matematis
Math.Sin(x)	Menghitung sinus dari sudut x (dalam radian).	$\sin(x)$
Math.Cos(x)	Menghitung kosinus dari sudut x (dalam radian).	$\cos(x)$
Math.Tan(x)	Menghitung tangen dari sudut x (dalam radian).	$\tan(x)$
Math.Asin(x)	Arcus sinus (kebalikan Sin), menghasilkan radian.	$\arcsin(x)$
Math.Atan2(y, x)	Menghitung sudut antara sumbu X positif dan titik (x, y) .	$\operatorname{atan2}(y, x)$

Konversi Derajat ke Radian

Karena Math.Sin() dkk menerima radian, Anda harus mengonversi derajat menggunakan rumus:

$$\text{Radian} = \text{Derajat} \times \left(\frac{\pi}{180}\right)$$

Contoh C#:



Shutterstock

C#

double derajat = 90;

double radian = derajat * (Math.PI / 180);

Console.WriteLine(Math.Sin(radian)); // Hasil: 1

2. Fungsi Eksponensial dan Logaritma

Fungsi ini digunakan untuk menghitung pangkat dan pertumbuhan alami.

Fungsi	Deskripsi	Rumus Matematis
Math.Pow(base, exp)	Menghitung angka pangkat tertentu.	x^y

Fungsi	Deskripsi	Rumus Matematis
Math.Exp(x)	Menghitung e^x (Bilangan Euler) pangkat x .	e^x
Math.Log(x)	Logaritma Natural (basis e).	$\ln(x)$
Math.Log10(x)	Logaritma basis 10.	$\log_{10}(x)$

Contoh C#:

C#

```
// Menghitung 2 pangkat 10
double pangkat = Math.Pow(2, 10); // 1024
```

```
// Menghitung Logaritma basis 10 dari 100
double log = Math.Log10(100); // 2
```

3. Fungsi Pembulatan Tambahan

Selain `Math.Round()`, ada dua fungsi pembulatan yang sangat spesifik dan sering digunakan untuk logika sistem (seperti pembagian halaman/pagination):

- **Math.Ceiling(x)**: Pembulatan ke **atas** (langit-langit). Selalu mengambil bilangan bulat terbesar berikutnya.
 - `Math.Ceiling(10.1)` $\rightarrow$ **11**
- **Math.Floor(x)**: Pembulatan ke **bawah** (lantai). Selalu membuang desimalnya.
 - `Math.Floor(10.9)` $\rightarrow$ **10**

Studi Kasus: Menghitung Tinggi Gedung

Jika Anda berdiri 10 meter dari gedung dan melihat ke atas dengan sudut 30° , berapa tinggi gedung tersebut?

(Rumus: $\text{Tinggi} = \text{Jarak} \times \tan(\text{Sudut})$)

C#

```
double jarak = 10;
double sudutDerajat = 30;
double sudutRadian = sudutDerajat * (Math.PI / 180);
```

```
double tinggi = jarak * Math.Tan(sudutRadian);
```

```
Console.WriteLine($"Tinggi gedung: {Math.Round(tinggi, 2)} meter");
```

```
// Output: Tinggi gedung: 5.77 meter
```

Apakah ada bagian dari kalkulasi matematika ini yang ingin Anda bedah lebih dalam, atau Anda ingin lanjut ke topik pemrograman C# lainnya?

- **9.4 Generate Random Numbers dengan `Random` Class**

Dalam C#, untuk menghasilkan angka acak, kita menggunakan class `Random`. Penting untuk dipahami bahwa angka yang dihasilkan sebenarnya adalah **pseudo-random**, yang berarti angka-angka tersebut dihasilkan melalui rumus matematika tertentu yang terlihat acak, tetapi sebenarnya memiliki pola jika kita mengetahui "seed" awalnya.

Berikut adalah panduan lengkap cara menggunakannya:

1. Inisialisasi Dasar

Untuk mulai menggunakan, Anda perlu membuat *instance* dari class `Random` terlebih dahulu.

C#

```
Random random = new Random();
```

2. Metode yang Sering Digunakan

Setelah membuat objek `random`, Anda bisa memanggil beberapa metode berikut:

Metode	Penjelasan	Contoh Hasil
Next()	Menghasilkan integer positif acak (0 sampai <code>int.MaxValue</code>).	1472839
Next(max)	Menghasilkan integer dari 0 sampai max - 1 .	<code>Next(10)</code> \$\\rightarrow\$ 0 s/d 9
Next(min, max)	Menghasilkan integer dari min sampai max - 1 .	<code>Next(1, 7)</code> \$\\rightarrow\$ dadu (1-6)
NextDouble()	Menghasilkan angka desimal (double) antara 0.0 dan 1.0 .	0.742...

3. Contoh Implementasi dalam Kode

C#

```
Random rng = new Random();
```

```
// 1. Membuat angka dadu (1 sampai 6)
```

```
int dadu = rng.Next(1, 7);
```

```
// 2. Membuat persentase acak (0.0 sampai 100.0)
```

```
double persen = rng.NextDouble() * 100;
```

```
// 3. Memilih elemen acak dari array
```

```
string[] hadiah = { "Kopi", "Teh", "Susu" };
```

```
int index = rng.Next(hadiah.Length);
```

```
string terpilih = hadiah[index];
```

```
Console.WriteLine($"Dadu: {dadu}, Hadiah: {terpilih}");
```

4. Perangkat Umum: Masalah "Seed"

Salah satu kesalahan pemula adalah membuat objek Random baru di dalam sebuah perulangan (*looping*).

Jangan lakukan ini:

C#

```
for (int i = 0; i < 5; i++) {
```

```
    Random r = new Random(); // Salah! Dibuat berulang kali dalam waktu sangat cepat
```

```
    Console.WriteLine(r.Next(1, 100));
```

}

Kenapa salah? Komputer bekerja sangat cepat. Karena Random secara default menggunakan waktu sistem sebagai "seed", jika objek dibuat dalam waktu yang hampir bersamaan, mereka akan menghasilkan urutan angka yang **sama persis**.

Solusinya: Buat satu objek Random di luar perulangan, lalu gunakan berulang kali.

5. Keamanan (Random vs RNGCryptoServiceProvider)

Jika Anda membutuhkan angka acak untuk keperluan **keamanan** (seperti membuat password atau token enkripsi), jangan gunakan Random. Gunakan RandomNumberGenerator dari System.Security.Cryptography.

- **Random:** Cepat, cocok untuk game dan simulasi.
 - **RandomNumberGenerator:** Lebih lambat, tapi tidak bisa ditebak polanya (aman secara kriptografi).
-



BAGIAN 2: STRUKTUR KONTROL DAN STRING

Bab 10: Manipulasi String

- **10.1 Deklarasi dan Inisialisasi String**
- Dalam C#, string adalah sekumpulan karakter yang digunakan untuk menyimpan teks. Secara teknis, string adalah objek dari class System.String, namun kita biasanya menggunakan kata kunci string (huruf kecil) sebagai alias yang lebih praktis.
- Berikut adalah cara mendeklarasikan dan menginisialisasi string dengan berbagai teknik:
- **1. Deklarasi dan Inisialisasi Dasar**
- Ada beberapa cara untuk membuat variabel string:
- C#
- // Deklarasi tanpa inisialisasi (masih bernilai null)
- `string pesan;`
-
- // Deklarasi dengan inisialisasi langsung
- `string sapaan = "Halo, selamat pagi!";`
-
- // Menggunakan kata kunci 'var' (Tipe data dideteksi otomatis)
- `var nama = "Gemini";`
-
- // Inisialisasi sebagai string kosong (Empty String)
- `string empty = "";`
- `string emptyModern = string.Empty; // Cara yang lebih disarankan (best practice)`

-
- - 2. Berbagai Jenis Literal String
 - C# menyediakan cara khusus untuk menulis teks, terutama jika teks tersebut mengandung karakter unik.
 - A. Verbatim String (@)
 - Digunakan jika Anda tidak ingin karakter *backslash* (\) dianggap sebagai perintah khusus (escape character). Sangat berguna untuk menulis alamat folder atau file.
 - C#
 - // Tanpa @ (harus pakai double backslash)
 - `string path1 = "C:\\Users\\Documents";`
 -
 - // Dengan @ (lebih bersih)
 - `string path2 = @"C:\Users\Documents";`
 - B. String Interpolation (\$)
 - Cara paling modern dan efisien untuk memasukkan variabel ke dalam teks.
 - C#
 - `string user = "Andi";`
 - `int umur = 25;`
 -
 - // Menggunakan tanda \$ dan kurung kurawal {}
 - `string info = $"User {user} berusia {umur} tahun.";`
 - C. Raw String Literals (""")
 - (Tersedia di C# 11+) Digunakan untuk menulis teks yang sangat panjang atau mengandung banyak tanda petik dan baris baru tanpa perlu ribet.
 - C#
 - `string json = """`
 - `{`
 - `"nama": "Andi",`
 - `"pekerjaan": "Programmer"`
 - `}`
 - `""";`
 -
-
- 3. Sifat String: Immutable (Tidak Dapat Diubah)
 - Satu hal penting yang perlu dipahami: **String di C# bersifat immutable.** Artinya, setelah string dibuat, isinya tidak bisa diubah di memori.
 - Jika Anda mencoba mengubahnya:
 - C#
 - `string teks = "Halo";`
 - `teks += " Dunia";`
 - C# sebenarnya tidak mengubah "Halo", melainkan menghapus "Halo" lama dan **membuat string baru** yang berisi "Halo Dunia".
 - **Tips:** Jika Anda perlu sering mengubah teks (seperti di dalam perulangan ribuan kali), gunakan class `StringBuilder` agar program lebih cepat dan hemat memori.
 -
-

- Perbandingan Karakter Khusus (Escape Sequences)
- Jika Anda menggunakan string biasa, gunakan karakter ini untuk memformat teks:

Kode	Hasil
\n	Baris Baru (New Line)
\t	Tab
\"	Menampilkan tanda petik dua
\\	Menampilkan backslash

- **10.2 Concatenation:** `+`, `String.Concat()`, `string.Join()`

Dalam C#, ada beberapa cara untuk menggabungkan string. Memilih metode yang tepat bergantung pada apakah Anda menggabungkan dua kata sederhana, sekumpulan data dari sebuah *list*, atau ingin mengoptimalkan penggunaan memori. Berikut adalah tiga cara utama untuk melakukan penggabungan (concatenation) di C#:

1. Operator Penjumlahan (+)

Ini adalah cara yang paling intuitif dan sering digunakan untuk penggabungan sederhana.

- **Cara Kerja:** Menggunakan simbol `+` di antara dua string atau lebih.
- **Kelebihan:** Sangat mudah dibaca untuk penggabungan yang sedikit.
- **Kekurangan:** Tidak efisien jika digunakan di dalam perulangan (*loop*) besar karena sifat string yang *immutable* (setiap `+` menciptakan objek baru di memori).

C#

```
string depan = "Budi";
```

```
string belakang = "Santoso";
```

```
string namaLengkap = depan + " " + belakang; // Hasil: "Budi Santoso"
```

2. String.Concat()

Metode ini secara teknis adalah apa yang dipanggil oleh compiler saat Anda menggunakan operator `+`. Namun, Anda bisa memanggilnya secara eksplisit.

- **Cara Kerja:** Menerima beberapa argumen string atau sebuah *array* string dan menyatukannya tanpa pemisah.
- **Kelebihan:** Lebih rapi jika Anda memiliki *array* yang ingin langsung disatukan menjadi satu string utuh.
- **Kekurangan:** Tidak menyediakan cara otomatis untuk menambahkan pemisah (seperti spasi atau koma) di antara elemen.

C#

```
string[] kata = { "Belajar ", "C# ", "Itu ", "Seru" };
```

```
string kalimat = string.Concat(kata); // Hasil: "Belajar C# Itu Seru"
```

3. String.Join()

Metode ini adalah yang paling "cerdas" jika Anda berurusan dengan daftar kata yang membutuhkan pemisah.

- **Cara Kerja:** Menerima karakter pemisah (separator) sebagai argumen pertama, lalu diikuti oleh kumpulan datanya.
- **Kelebihan:** Secara otomatis hanya menambahkan pemisah **di antara** kata-kata (tidak akan ada koma tambahan di akhir kalimat). Sangat efisien untuk memformat data dari *List* atau *Array*.

C#

```
string[] hobi = { "Coding", "Membaca", "Lari" };
```

```
// Menggabungkan dengan koma dan spasi
```

```
string daftarHobi = string.Join(", ", hobi);
```

```
Console.WriteLine(daftarHobi); // Hasil: "Coding, Membaca, Lari"
```

Perbandingan Cepat

Metode	Hasil Terbaik Untuk...	Contoh Penggunaan
Operator +	Menggabungkan 2-3 variabel string.	str1 + str2
String.Concat	Menyatukan array string tanpa jeda.	string.Concat(myArray)
String.Join	Membuat daftar yang dipisahkan simbol.	string.Join(" ", items)
Interpolation (\$)	Memasukkan variabel ke dalam kalimat.	\$"Nama saya {nama}"

Tips Pro: Kapan Menggunakan StringBuilder?

Jika Anda harus menggabungkan string di dalam sebuah perulangan (misalnya menggabungkan 1.000 kata dari database), jangan gunakan + atau string.Concat. Gunakanlah **StringBuilder**.

C#

```
using System.Text;
```

```
StringBuilder sb = new StringBuilder();
```

```
for (int i = 0; i < 100; i++) {
```

```
    sb.Append("Data-");
```

```
    sb.Append(i);
```

```
sb.Append(" ");
}
string hasilAkhir = sb.ToString();
StringBuilder jauh lebih hemat memori karena ia memodifikasi satu buffer yang sama
alih-alih terus-menerus membuat objek string baru.
```

- **10.3 String Interpolation dengan \$**

String Interpolation (ditandai dengan simbol \$) adalah cara paling modern, bersih, dan efisien untuk memasukkan nilai variabel atau ekspresi langsung ke dalam sebuah string di C#.

Sebelum adanya fitur ini (diperkenalkan pada C# 6), programmer harus menggunakan `string.Format()` atau operator `+` yang sering kali sulit dibaca jika variabelnya banyak.

1. Sintaks Dasar

Untuk menggunakan interpolation, Anda harus meletakkan tanda \$ tepat sebelum tanda kutip pembuka. Variabel dimasukkan ke dalam kurung kurawal {}.

C#

```
string nama = "Gemini";
int versi = 3;
```

```
// Menggunakan String Interpolation
```

```
string pesan = $"Halo, saya {nama} versi {versi}.";
```

```
Console.WriteLine(pesan);
```

```
// Output: Halo, saya Gemini versi 3.
```

2. Menjalankan Ekspresi di Dalam {}

Hebatnya, Anda tidak hanya bisa memasukkan variabel, tetapi juga operasi matematika, pemanggilan fungsi, atau logika ternary di dalam kurung kurawal tersebut.

C#

```
int angka1 = 10;
int angka2 = 20;
```

```
Console.WriteLine($"Hasil penjumlahannya adalah: {angka1 + angka2}");
```

```
// Output: Hasil penjumlahannya adalah: 30
```

```
Console.WriteLine($"Status: {(angka1 > 5 ? "Besar" : "Kecil")}");
```

```
// Output: Status: Besar
```

3. Pemformatan Angka dan Tanggal

Interpolation juga memudahkan Anda mengatur tampilan angka (seperti mata uang atau desimal) dan tanggal tanpa perlu fungsi tambahan.

Format	Kegunaan	Contoh Kode	Hasil (ID-id)
:C	Currency (Mata Uang)	\$"Harga: {5000:C}"	Rp5.000,00
:N2	Number (2 Desimal)	\$"Nilai: {3.14159:N2}"	3,14
:P	Percent (Persen)	\$"Progress: {0.75:P}"	75,00%
:yyyy	Format Tahun	\$"Tahun: {DateTime.Now:yyyy}"	2026

4. Kombinasi dengan Verbatim String (\$@)

Jika Anda ingin membuat string yang memiliki banyak baris (multiline) atau alamat path folder, tetapi tetap ingin memasukkan variabel, Anda bisa menggabungkan \$ dan @.

C#

```
string folder = "DataUser";
string query = $"@\"C:\Project\{folder}\config.json\";
```

```
string surat = $"@\"
    Halo {nama},
    Tagihan Anda bulan ini adalah {250000:C}.
    Terima kasih.\";
```

● 10.4 Mengakses Karakter dalam String

Dalam C#, `string` pada dasarnya adalah **array dari karakter** (`char`). Karena itu, cara mengaksesnya sangat mirip dengan cara kita mengakses elemen dalam sebuah array, yaitu menggunakan **indeks**.

Berikut adalah teknik-teknik untuk mengakses karakter dalam string:

1. Menggunakan Indeks (□)

Setiap karakter dalam string memiliki nomor posisi yang disebut indeks, dimulai dari **0** untuk karakter pertama.

C#

- `string` kata = "Belajar";
-
- `char` hurufPertama = kata[0]; // 'B'
- `char` hurufKetiga = kata[2]; // 'l'

- - `Console.WriteLine($"Huruf pertama adalah {hurufPertama}");`
 - **Peringatan:** String di C# bersifat *read-only* jika diakses via indeks. Anda bisa membaca `kata[0]`, tapi Anda **tidak bisa** mengubahnya seperti `kata[0] = 'b'`. Untuk mengubahnya, Anda harus menggunakan metode seperti `Replace` atau `ToCharArray`.
-

• 2. Mengakses Karakter Terakhir

- Untuk mendapatkan karakter terakhir tanpa menghitung manual, kita gunakan properti `.Length`. Karena indeks dimulai dari 0, maka indeks terakhir adalah **panjang string dikurangi 1**.
 - **Cara Tradisional:**
 - C#
 - `string teks = "Programmer";`
 - `char terakhir = teks[teks.Length - 1]; // 'r'`
 - **Cara Modern (C# 8.0+):**
 - C# memperkenalkan operator "hat" (^) untuk menghitung dari belakang.
 - C#
 - `char terakhirModern = teks[^1]; // Karakter terakhir`
 - `char keduaTerakhir = teks[^2]; // 'e'`
-

• 3. Iterasi (Mengakses Semua Karakter)

- Jika Anda ingin mengakses setiap karakter satu per satu (misalnya untuk menyaring huruf vokal), Anda bisa menggunakan perulangan.
 - Menggunakan `foreach` (Paling Sempel):
 - C#
 - `string nama = "C#";`
 - `foreach (char c in nama)`
 - `{`
 - `Console.WriteLine(c);`
 - `}`
 - Menggunakan `for` (Jika butuh indeks):
 - C#
 - `for (int i = 0; i < nama.Length; i++)`
 - `{`
 - `Console.WriteLine($"Indeks ke-{i} adalah {nama[i]}");`
 - `}`
-

• 4. Mengecek Keberadaan Karakter

- Jika Anda hanya ingin tahu apakah suatu karakter ada di dalam string atau di posisi mana, gunakan metode bawaan ini:

Metode	Kegunaan	Contoh
Contains()	Cek apakah karakter/teks ada.	teks.Contains('a') $\rightarrow$ true/false
IndexOf()	Mencari indeks pertama karakter tersebut.	teks.IndexOf('r') $\rightarrow$ 1
LastIndexOf()	Mencari indeks terakhir karakter tersebut.	teks.LastIndexOf('r') $\rightarrow$ 9

- **10.5 Metode String Populer:** `Length`, `ToUpper()`, `ToLower()`, `Trim()`, `Substring()`, `Replace()`, `Split()`

C# menyediakan berbagai metode bawaan dalam class String yang mempermudah kita untuk memanipulasi teks tanpa harus memproses karakter satu per satu secara manual.

Berikut adalah daftar metode string yang paling sering digunakan dalam pengembangan aplikasi:

1. Length (Properti)

Meskipun teknisnya adalah properti (bukan metode), Length sangat penting untuk mengetahui jumlah total karakter dalam string, termasuk spasi dan tanda baca.

- **Hasil:** int
- **Contoh:** "Halo".Length $\rightarrow$ 4

2. ToUpper() dan ToLower()

Digunakan untuk mengubah standarisasi huruf. Sangat berguna saat membandingkan string (misalnya email) agar tidak terpengaruh oleh huruf besar/kecil (*case-insensitive*).

- **ToUpper():** Mengubah semua huruf menjadi kapital.
- **ToLower():** Mengubah semua huruf menjadi huruf kecil.
- **Contoh:** "Csharp".ToUpper() $\rightarrow$ "CSHARP"

3. Trim(), TrimStart(), TrimEnd()

Digunakan untuk membersihkan spasi kosong (atau karakter tertentu) di bagian awal dan akhir string. Biasanya digunakan untuk membersihkan input dari user.

- **Contoh:** " Halo ".Trim() $\rightarrow$ "Halo"

4. Substring()

Digunakan untuk mengambil sebagian teks dari string utama berdasarkan indeks tertentu.

- **Sintaks:** Substring(indeksMulai, panjang)
- **Contoh:** ``csharp

```
string teks = "Belajar C#";
string hasil = teks.Substring(0, 7); // Mengambil 7 karakter mulai dari indeks 0
// Hasil: "Belajar"
```

5. Replace()

Digunakan untuk mengganti karakter atau sekumpulan teks tertentu dengan teks yang baru.

- **Sintaks:** Replace(teksLama, teksBaru)
- **Contoh:** "Saya suka Java".Replace("Java", "C#") $\rightarrow$ "Saya suka C#"

6. Split()

Memotong string menjadi sebuah **Array** berdasarkan karakter pemisah (delimiter) tertentu. Sangat berguna untuk memproses data CSV atau kalimat.

- **Contoh:**
C#
string hobi = "Coding,Membaca,Lari";
string[] kumpulanHobi = hobi.Split(',');
// Hasil: ["Coding", "Membaca", "Lari"]

Tabel Ringkasan Metode String

Metode	Input	Contoh Hasil	Kegunaan Utama
Length	"Budi"	4	Menghitung jumlah karakter.
ToUpper()	"a"	"A"	Konversi ke huruf besar.
Trim()	" x "	"x"	Menghapus spasi di ujung-ujung.
Substring()	(0, 2)	"Ba" (dari "Baru")	Mengambil potongan teks.
Replace()	("A", "B")	"B" (dari "A")	Mengganti isi teks.
Split()	','	string[]	Memecah string menjadi array.

Contoh Implementasi Gabungan

Mari kita lihat bagaimana metode-metode ini bekerja sama untuk membersihkan data input:

```
1 string inputUser = " PROGRAMMER C# ";
2
3 // 1. Bersihkan spasi dan buat huruf kecil semua
4 string clean = inputUser.Trim().ToLower(); // "programmer c#"
5
6 // 2. Ganti spasi dengan underscore
7 string slug = clean.Replace(" ", "_"); // "programmer_c#"
8
9 // 3. Ambil 10 karakter pertama
10 string singkat = slug.Substring(0, 10); // "programmer"
11
12 Console.WriteLine(singkat);
13
```

- **10.6 String vs StringBuilder: Kapan Menggunakan?**

Ini adalah pertanyaan klasik yang membedakan programmer C# pemula dengan yang sudah berpengalaman. Kunci perbedaannya terletak pada **Manajemen Memori**.

- **1. String: Si "Immutable"**

Seperti yang sudah kita bahas, string bersifat **Immutable** (tidak dapat diubah). Setiap kali Anda melakukan operasi seperti +, Replace(), atau ToUpper(), C# tidak mengubah string yang asli. Sebaliknya, ia membuang string lama dan mengalokasikan ruang baru di memori (Heap) untuk string yang baru.

- **Contoh Buruk:**

```
C#
string hasil = "";
for (int i = 0; i < 1000; i++) {
    hasil += i.ToString(); // C# membuat 1000 objek string baru di memori!
}
```

Analogi: Seperti menulis surat, tapi setiap kali ingin menambah satu kata, Anda harus membuang kertas lama dan menyalin semuanya ke kertas baru yang lebih besar.

- **2. StringBuilder: Si "Mutable"**

StringBuilder (berada dalam *namespace* System.Text) bersifat **Mutable** (dapat diubah). Ia bekerja dengan menyediakan sebuah wadah (*buffer*) di memori yang bisa diperbesar secara dinamis.

- **Cara Kerja:** Ia memodifikasi konten di lokasi memori yang sama tanpa menciptakan objek baru terus-menerus.

- **Contoh Baik:**

```
1 using System.Text;
2
3 StringBuilder sb = new StringBuilder();
4 for (int i = 0; i < 1000; i++)
5 {
6     sb.Append(i); // Hanya memodifikasi satu objek yang sama
7 }
8 string hasilAkhir = sb.ToString(); // Diubah ke string hanya di akhir
9
```

- **3. Kapan Menggunakan yang Mana?**

Gunakan String Jika...	Gunakan StringBuilder Jika...
Jumlah modifikasi teks sedikit (1-5 kali).	Melakukan modifikasi di dalam Perulangan (Loop) .
Anda hanya melakukan operasi sederhana seperti + pada 2-3 variabel.	Anda sedang memproses data teks yang sangat besar (misal: parsing file log).
Keterbacaan kode lebih penting daripada performa ekstrem.	Anda perlu membangun string secara dinamis berdasarkan banyak kondisi.
Anda sering melakukan pencarian seperti Contains() atau IndexOf().	Anda sering melakukan penghapusan atau penyisipan teks di tengah (Insert, Remove).

Kesimpulan Praktis

- Untuk **konstanta** atau teks statis: Pakai string.
- Untuk **manipulasi berat**: Pakai StringBuilder.

Fun Fact: Sebenarnya, jika Anda hanya menggabungkan string dalam satu baris seperti `string s = "a" + "b" + "c" + "d";`, Compiler C# cukup pintar untuk mengubahnya menjadi `string.Concat` secara otomatis, sehingga performanya tetap baik. Namun, jika penggabungan itu terjadi di langkah yang terpisah (terutama dalam *loop*), `StringBuilder` adalah pemenang mutlak.

• 10.7 Escape Characters dan Verbatim Strings (@)

Dalam C#, **Escape Characters** dan **Verbatim Strings** adalah dua cara berbeda untuk menangani karakter-karakter "sulit" di dalam string, seperti tanda petik, *backslash*, atau baris baru.

1. Escape Characters (Karakter Pelarian)

Escape characters menggunakan tanda *backslash* (\) diikuti oleh karakter tertentu untuk memberikan instruksi khusus kepada compiler.

Kode	Deskripsi	Hasil
\n	Newline	Membuat baris baru.
\t	Horizontal Tab	Memberikan jarak tab (indentasi).
\"	Double Quote	Menampilkan tanda petik dua di dalam teks.
\\	Backslash	Menampilkan satu karakter backslash.
\'	Single Quote	Menampilkan tanda petik satu (biasanya untuk char).

Contoh Kode:

```

1  string teks = "Baris satu\nBaris dua";
2  string kutipan = "Dia berkata, \"Halo C#\"";
3  string path = "C:\\Users\\Admin"; // Perlu double backslash
4

```

2. Verbatim Strings (@)

Verbatim string diawali dengan simbol @ sebelum tanda kutip pembuka. Fitur ini memerintahkan C# untuk membaca string **apa adanya** tanpa mempedulikan escape characters (kecuali tanda petik dua).

Kegunaan Utama:

- **Menulis Path Folder:** Tidak perlu lagi menulis \\.
- **Multiline String:** Anda bisa langsung menekan *Enter* di dalam kode untuk membuat baris baru.

Contoh Kode:

```

1 // Path folder jadi lebih manusiawi
2 string pathData = @"C:\Users\Documents\Project";
3
4 // Membuat teks berbaris-baris (Multiline)
5 string puisi = @"
6     Matahari terbenam,
7     Hari mulai malam,
8     C# tetap asik dipelajari.";
9
10 // Menampilkan tanda petik di Verbatim (Gunakan double quote "")
11 string petik = @"Dia berkata ""Halo"" kepada saya.";
12

```

Perbandingan Visual

Fitur	String Biasa	Verbatim String (@)
Backslash	Harus di-escape (\)	Ditulis apa adanya (\)
Baris Baru	Harus pakai \n	Cukup tekan Enter
Tanda Petik	Pakai \"	Pakai tanda petik ganda ""
Keterbacaan	Agak berantakan untuk path	Sangat bersih dan rapi

Bab 11: Boolean dan Logika Kondisional

• 11.1 Tipe Data `bool` dan Penggunaannya

Dalam C#, tipe data **bool** (singkatan dari *Boolean*) adalah tipe data paling sederhana namun paling krusial untuk logika program. Variabel bool hanya bisa menampung dua nilai: **true** (benar) atau **false** (salah).

1. Deklarasi dan Inisialisasi

Secara default, jika Anda mendeklarasikan bool sebagai *field* di dalam class tanpa memberikan nilai, maka nilai defaultnya adalah **false**.

```
1 bool isReady = true;
2 bool isFinished = false;
3 var isLogged = true; // Compiler otomatis mendeteksi sebagai bool
4
```

2. Hasil dari Operator Perbandingan

Tipe data bool seringkali tidak ditulis manual, melainkan dihasilkan dari ekspresi perbandingan.

```
1 int nilai = 80;
2 bool lulus = (nilai >= 75); // Akan menghasilkan true
3
4 Console.WriteLine(lulus); // Output: True
5
```

3. Penggunaan dalam Struktur Kontrol

bool adalah "bahan bakar" utama untuk perintah percabangan (if) dan perulangan (while).

A. Pernyataan if

```
1 bool sedangHujan = true;
2
3 if (sedangHujan)
4 {
5     Console.WriteLine("Bawa payung!");
6 }
7
```

B. Perulangan while

```
1 bool lanjutMain = true;
2 int nyawa = 3;
3
4 while (lanjutMain)
5 {
6     Console.WriteLine("Sedang bermain...");
7     nyawa--;
8
9     if (nyawa <= 0)
10    {
11        lanjutMain = false; // Menghentikan loop
12    }
13 }
14
```

4. Operator Logika pada Boolean

Anda bisa menggabungkan atau memanipulasi nilai bool menggunakan operator logika:

- **&& (AND)**: Benar jika keduanya benar.

- **|| (OR):** Benar jika salah satu benar.
- **! (NOT):** Membalikkan nilai (true jadi false).

```

1  bool punyaKunci = true;
2  bool tahuPIN = false;
3
4  // Harus punya kunci DAN tahu PIN
5  bool bolehMasuk = punyaKunci && tahuPIN; // false
6
7

```

5. Praktis: Penamaan Variabel Boolean

Dalam dunia profesional, variabel bool biasanya dinamai dengan awalan kata kerja pembantu seperti **is**, **has**, **can**, atau **should**. Ini membuat kode Anda terbaca seperti bahasa Inggris manusia.

- `isEmailValid` (Apakah email valid?)
- `hasGoldMember` (Apakah punya member emas?)
- `canDeleteRecord` (Dapatkah menghapus data?)

6. Nullable Boolean (bool?)

Terkadang, Anda butuh status ketiga selain Benar dan Salah, yaitu **"Tidak Tahu"** atau **"Kosong"**. Di C#, Anda bisa menggunakan *Nullable* bool.

```

1  bool? isMember = null; // Bisa bernilai true, false, atau null
2
3  if (isMember == null)
4  {
5      Console.WriteLine("Status member tidak diketahui.");
6  }
7

```

• 11.2 Ekspresi Boolean dalam Kondisi

Dalam C#, **Ekspresi Boolean** adalah kombinasi dari variabel, nilai, dan operator yang ketika dievaluasi akan menghasilkan satu nilai: true atau false. Ekspresi inilah yang menjadi "otak" di balik setiap struktur kendali seperti `if`, `while`, dan `switch`. Berikut adalah cara membangun dan menggunakan ekspresi boolean secara efektif:

1. Struktur Dasar dalam Kondisi

Pernyataan `if` hanya akan mengeksekusi blok kode di dalamnya jika ekspresi di dalam kurung bernilai true.

```

1  int stok = 10;
2  bool adaBarang = (stok > 0); // Ini adalah ekspresi boolean
3
4  if (adaBarang)
5  {
6      Console.WriteLine("Silakan beli.");
7  }
8

```

2. Menggabungkan Ekspresi Kompleks

Seringkali satu syarat tidak cukup. Kita menggunakan operator logika (&&, ||, !) untuk menggabungkan beberapa ekspresi.

- **AND (&&):** Semua syarat wajib terpenuhi.
- **OR (||):** Minimal satu syarat terpenuhi.
- **NOT (!):** Membalikkan logika.

```

1  int umur = 20;
2  bool punyaKTP = true;
3  bool isBanned = false;
4
5  // Ekspresi: Umur >= 17 DAN punya KTP DAN TIDAK sedang di-banned
6  if (umur >= 17 && punyaKTP && !isBanned)
7  {
8      Console.WriteLine("Akses diberikan.");
9  }
10

```

3. Pembersihan Kode (Boolean Clean Code)

Banyak pemula menulis ekspresi boolean secara berlebihan. Berikut cara menulisnya seperti profesional:

Cara Pemula (Redundan)	Cara Profesional (Clean)
if (isLunas == true)	if (isLunas)
if (isLunas == false)	if (!isLunas)
bool cek = x > 10 ? true : false;	bool cek = x > 10;

4. Pattern Matching (C# 9.0+)

C# modern memperkenalkan cara yang lebih manusiawi untuk menulis ekspresi boolean yang kompleks menggunakan kata kunci `is`, `and`, `or`, dan `not`.

```

1  int temperatur = 25;
2
3  // Memeriksa rentang angka dengan gaya bahasa manusia
4  if (temperatur is > 20 and < 30)
5  {
6      Console.WriteLine("Cuaca nyaman.");
7  }
8
9  object data = "Halo";
10 if (data is not int)
11 {
12     Console.WriteLine("Ini bukan angka.");
13 }
14

```

5. Short-Circuit Evaluation

Penting untuk diingat bahwa C# sangat efisien.

- Pada &&, jika ekspresi pertama false, C# **tidak akan mengecek** sisanya karena hasilnya pasti false.
- Pada ||, jika ekspresi pertama true, C# **tidak akan mengecek** sisanya karena hasilnya pasti true.

Contoh Keamanan:

```
1 string teks = null;
2
3 // C# tidak akan error karena teks != null bernilai false,
4 // sehingga teks.Length tidak akan pernah dipanggil.
5 if (teks != null && teks.Length > 0)
6 {
7     Console.WriteLine("Ada isinya.");
8 }
```

11.3 Operator Logika Lanjutan

Setelah menguasai dasar-dasar &&, ||, dan !, saatnya kita masuk ke level lebih dalam. Dalam C#, terdapat beberapa operator logika lanjutan yang sering digunakan untuk optimasi performa dan manipulasi data pada level yang lebih rendah.

Berikut adalah rinciannya:

1. Conditional Logical vs. Logical Biasa

Banyak yang tidak menyadari bahwa C# memiliki dua versi operator AND dan OR.

Nama	Operator	Perilaku
Conditional (Short-circuit)	&& dan `	
Logical (Non-short-circuit)	& dan `	`

Kapan menggunakan & atau | pada Boolean?

Gunakan ini jika Anda ingin sebuah fungsi di sisi kanan **tetap berjalan** meskipun sisi kiri sudah menentukan hasil akhirnya.

```
1 bool LogAction()
2 {
3     Console.WriteLine("Aksi dicatat!");
4     return true;
5 }
6
7 bool test = false && LogAction(); // LogAction TIDAK akan jalan (Short-circuit)
8 bool test2 = false & LogAction(); // LogAction TETAP jalan (Non-short-circuit)
9
```

2. Operator Logical XOR (^)

Operator **XOR** (*Exclusive OR*) akan menghasilkan true jika **dan hanya jika** kedua operan berbeda (satu true, satu false).

- **Rumus:** $\text{true} \wedge \text{true} = \text{false}$, $\text{false} \wedge \text{false} = \text{false}$, $\text{true} \wedge \text{false} = \text{true}$.
- **Analogi:** "Anda bisa memilih Kopi atau Teh, tapi tidak boleh keduanya dan tidak boleh tidak memilih."

```
1 bool punyaKopi = true;
2 bool punyaTeh = false;
3
4 bool pilihSatu = punyaKopi ^ punyaTeh; // Hasil: true
5
```

3. Logika "Is" dan "Not" (Pattern Matching)

Di versi C# terbaru (9.0+), kita bisa menulis ekspresi logika yang jauh lebih mirip dengan bahasa manusia. Ini membuat kode lebih mudah dibaca dan mengurangi kesalahan logika.

- **and:** Pengganti && dalam pola.
- **or:** Pengganti || dalam pola.
- **not:** Pengganti ! dalam pola.

```
1 int suhu = 25;
2
3 // Jauh lebih bersih daripada (suhu > 20 && suhu < 30)
4 if (suhu is > 20 and < 30)
5 {
6     Console.WriteLine("Suhu ideal.");
7 }
8
9 object nama = null;
10 if (nama is not string)
11 {
12     Console.WriteLine("Ini bukan teks.");
13 }
```

4. Operator Logika Majemuk (Assignment)

Sama seperti operator aritmatika (+=), operator logika juga punya bentuk penugasan singkat. Ini sangat berguna saat mengelola *state* atau *flag* dalam program.

- **&=:** isAktif &= isValid (sama dengan isAktif = isAktif & isValid)
- **|=:** isAktif |= isValid (sama dengan isAktif = isAktif | isValid)
- **^=:** isAktif ^= true (Toggle/Saklar: membalikkan nilai true ke false dan sebaliknya)

Ringkasan Strategi Logika

1. **Gunakan && dan ||** untuk efisiensi (hampir 99% kasus).
2. **Gunakan ^** untuk logika "salah satu saja".
3. **Gunakan and, or, not** untuk pengecekan tipe data atau rentang angka agar kode lebih "pro".
4. **Hati-hati dengan & dan |**; pastikan Anda memang ingin sisi kanan dieksekusi sebelum menggunakannya.

Bab 12: Percabangan: If, Else, dan Switch

- **12.1 Struktur if, else if, else**

Struktur percabangan if, else if, dan else adalah fondasi dari logika pengambilan keputusan dalam C#. Struktur ini memungkinkan program Anda untuk menjalankan blok kode yang berbeda berdasarkan kondisi tertentu.

1. Struktur Dasar if

Blok if adalah perintah paling simpel. Kode di dalamnya hanya akan berjalan jika kondisinya bernilai true.

```
1      int umur = 20;
2
3      if (umur >= 18)
4      {
5          Console.WriteLine("Anda sudah dewasa.");
6      }
```

2. Menambahkan else

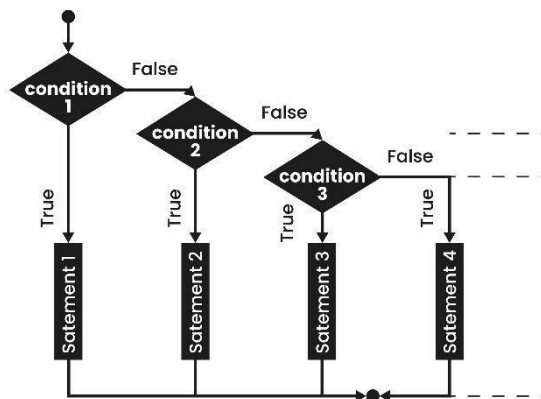
else digunakan sebagai "rencana cadangan". Jika kondisi pada if bernilai false, maka blok kode di dalam else akan dijalankan.

```
1      int nilai = 65;
2
3      if (nilai >= 75)
4      {
5          Console.WriteLine("Lulus");
6      }
7      else
8      {
9          Console.WriteLine("Tidak Lulus");
10     }
```

3. Struktur Lengkap: if, else if, dan else

Jika Anda memiliki lebih dari dua kemungkinan, gunakan else if. Anda bisa menambahkan else if sebanyak yang dibutuhkan, namun hanya satu blok yang akan dieksekusi (yang pertama kali memenuhi syarat).

IF-ELSE-IF STATEMENT



```
1      int suhu = 30;
2
3      if (suhu > 35)
4      {
5          Console.WriteLine("Cuaca sangat panas!");
6      }
7      else if (suhu >= 25 && suhu <= 35)
8      {
9          Console.WriteLine("Cuaca hangat.");
10     }
11     else if (suhu >= 15 && suhu < 25)
12     {
13         Console.WriteLine("Cuaca sejuk.");
14     }
15     else
16     {
17         Console.WriteLine("Cuaca dingin.");
18     }
```

4. Hal-Hal Penting untuk Diingat

- **Urutan itu Penting:** C# memeriksa kondisi dari atas ke bawah. Begitu satu kondisi terpenuhi, kondisi di bawahnya tidak akan diperiksa lagi.
- **Blok else bersifat Opsional:** Anda tidak wajib menggunakan else.
- **Tanpa Kurung Kurawal { }:** Jika kode di dalam blok hanya **satu baris**, Anda boleh meniadakan { }. Namun, sangat disarankan untuk tetap menggunakannya demi keterbacaan dan menghindari bug di masa depan.

```
1 // Boleh, tapi kurang disarankan
2 if (isReady) Console.WriteLine("Ready!");
3
```

5. Nested if (if bersarang)

Anda bisa memasukkan struktur if di dalam if lainnya untuk logika yang lebih mendalam.

```
1 bool punyaKunci = true;
2 bool alarmMati = false;
3
4 if (punyaKunci)
5 {
6     if (alarmMati)
7     {
8         Console.WriteLine("Pintu terbuka, silakan masuk.");
9     }
10    else
11    {
12        Console.WriteLine("Kunci ada, tapi alarm masih bunyi!");
13    }
14 }
```

Tips Pro: Pilih if-else atau switch?

- Gunakan **if-else** jika Anda berurusan dengan rentang angka (misal: suhu > 30) atau logika boolean yang kompleks.
- Gunakan **switch** jika Anda hanya ingin membandingkan satu variabel dengan banyak nilai tetap (misal: mengecek nama hari atau kode menu).

- **12.2 Nested If dan Best Practices**

Nested If adalah kondisi di mana Anda menempatkan satu struktur if di dalam if lainnya. Meskipun sangat berguna untuk mengecek kondisi yang berjenjang, penggunaan yang berlebihan bisa menyebabkan kode menjadi sulit dibaca, yang sering disebut sebagai "**Pyramid of Doom**" (kode yang menjorok terlalu jauh ke kanan).

Berikut adalah panduan cara kerja dan cara mengoptimalkannya menggunakan *Best Practices*.

1. Contoh Dasar Nested If

Bayangkan sebuah sistem login: Pertama kita cek apakah emailnya ada, jika ada, baru kita cek apakah passwordnya benar.

```

1  string emailInput = "user@mail.com";
2  string passInput = "12345";
3
4  if (emailInput == "user@mail.com")
5  {
6      if (passInput == "12345")
7      {
8          Console.WriteLine("Login Berhasil.");
9      }
10     else
11     {
12         Console.WriteLine("Password Salah.");
13     }
14 }
15 else
16 {
17     Console.WriteLine("Email tidak terdaftar.");
18 }
19

```

2. Best Practices (Praktik Terbaik)

Agar kode tidak menjadi berantakan, para profesional biasanya menggunakan teknik berikut:

A. Gunakan Guard Clauses (Pola "Fail First")

Alih-alih membungkus kode utama di dalam banyak if, lebih baik kita "mengusir" kondisi yang salah di awal menggunakan `return` atau `continue`. Ini membuat kode tetap rata di sebelah kiri.

B. Gabungkan Kondisi dengan Operator Logika

Jika dua kondisi harus terpenuhi secara bersamaan, jangan gunakan nested if. Gunakan operator `&&`.

C. Batasi Kedalaman (Maksimal 2-3 Level)

Jika Anda merasa sudah masuk ke level ke-4 dari nested if, itu tandanya kode Anda perlu di-**Refactor**. Pecahlah logika tersebut menjadi fungsi/method baru yang lebih kecil.

3. Perbandingan Visual

Teknik	Kelebihan	Kekurangan
Nested If	Alur logika terlihat berjenjang.	Cepat membuat kode berantakan dan sulit di-debug.
Guard Clauses	Sangat bersih, mudah dibaca, "Happy Path" terlihat jelas.	Membutuhkan penggunaan <code>return</code> , <code>break</code> , atau <code>throw</code> yang tepat.

Teknik	Kelebihan	Kekurangan
Logical Operators	Ringkas dan to-the-point.	Bisa menjadi panjang jika kondisinya terlalu banyak.

• 12.3 Ternary Operator untuk Kondisi Sederhana

Operator Ternary (?:) adalah versi "diet" atau bentuk singkat dari struktur if-else. Operator ini sangat efektif untuk menetapkan nilai ke variabel berdasarkan suatu kondisi hanya dalam **satu baris kode**.

Disebut "ternary" karena melibatkan tiga bagian: **Kondisi**, **Hasil jika benar**, dan **Hasil jika salah**.

1. Sintaks dan Cara Kerja

Sintaksnya mengikuti pola berikut:

```
variabel = (kondisi) ? nilai_jika_true : nilai_jika_false;
```

Contoh Sederhana:

Bayangkan Anda ingin menentukan apakah seseorang boleh masuk ke bioskop berdasarkan umurnya.

Versi if-else (4 Baris):

```
C#
string status;
if (umur >= 17) {
    status = "Boleh Masuk";
} else {
    status = "Dilarang Masuk";
}
```

Versi Ternary (1 Baris):

```
C#
string status = (umur >= 17) ? "Boleh Masuk" : "Dilarang Masuk";
```

2. Kapan Harus Menggunakan Ternary?

Ternary bukan pengganti if-else untuk semua situasi. Gunakanlah jika:

- Anda ingin melakukan **penugasan nilai** (assignment).
- Kondisinya **sederhana** dan pendek.
- Logikanya tidak memiliki banyak instruksi (hanya mengembalikan nilai).

3. Ternary di Dalam String Interpolation

Salah satu kekuatan utama operator ini adalah kemampuannya "nyelip" di dalam teks (string interpolation), sehingga Anda tidak perlu membuat variabel perantara.

```
C#
int stok = 5;
Console.WriteLine($"Status barang: {(stok > 0 ? "Tersedia" : "Habis")}");
```

4. Nesting Ternary (Ternary Bersarang)

Anda bisa memasukkan ternary di dalam ternary lainnya, tapi **hati-hati**, ini bisa membuat kode sulit dibaca jika terlalu panjang.

C#

```
int nilai = 85;
```

```
string grade = (nilai >= 90) ? "A" : (nilai >= 80) ? "B" : "C";
```

Tips: Jika sudah mencapai 3 kondisi atau lebih, beralihlah kembali ke if-else if atau switch agar rekan tim (atau Anda di masa depan) tidak pusing membacanya.

Perbandingan Cepat

Fitur	if-else	Ternary (?:)
Jumlah Baris	Lebih banyak	Sangat ringkas (1 baris)
Kegunaan	Logika kompleks & banyak aksi	Penugasan nilai sederhana
Keterbacaan	Sangat jelas untuk pemula	Elegan tapi bisa rumit jika di-nest
Output	Menjalankan blok kode	Mengembalikan sebuah nilai

Kesalahan Umum

Anda tidak bisa menggunakan ternary sebagai pengganti perintah yang tidak mengembalikan nilai (void).

- **Salah:** (umur > 17) ? Console.WriteLine("A") : Console.WriteLine("B");
- **Benar:** Console.WriteLine(umur > 17 ? "A" : "B");

- **12.4 Statement `switch` dan `case`**

Struktur switch adalah alternatif yang jauh lebih rapi daripada menggunakan rangkaian if-else if yang terlalu panjang. switch sangat ideal ketika Anda ingin membandingkan **satu variabel** dengan **banyak nilai tetap** (konstanta).

1. Sintaks Dasar switch

Bayangkan Anda membuat sistem menu sederhana. Daripada mengecek satu per satu, switch langsung melompat ke nilai yang cocok.

```

1  int pilihan = 2;
2
3  switch (pilihan)
4  {
5      case 1:
6          Console.WriteLine("Membuka Menu Profil");
7          break;
8      case 2:
9          Console.WriteLine("Membuka Pengaturan");
10         break;
11     case 3:
12         Console.WriteLine("Keluar dari Aplikasi");
13         break;
14     default:
15         Console.WriteLine("Pilihan tidak valid.");
16         break;
17 }

```

2. Komponen Penting

- **case:** Nilai yang ingin dibandingkan. Harus berupa konstanta (angka, string, atau enum).
- **break: Wajib ada.** Ini digunakan untuk menghentikan eksekusi dan keluar dari blok switch. Tanpa break, C# akan memberikan error (kecuali jika case tersebut kosong).
- **default:** Bersifat opsional, tapi sangat disarankan. Ini akan dijalankan jika tidak ada satu pun case yang cocok (mirip seperti else).

3. Menggabungkan Case (Stacking)

Jika beberapa nilai menghasilkan tindakan yang sama, Anda bisa menumpuk case tanpa memberikan break di antaranya.

```

1  char grade = 'B';
2
3  switch (grade)
4  {
5      case 'A':
6      case 'B':
7      case 'C':
8          Console.WriteLine("Anda Lulus!");
9          break;
10     case 'D':
11     case 'E':
12         Console.WriteLine("Anda Tidak Lulus.");
13         break;
14 }

```

4. Switch Modern: Switch Expression (C# 8.0+)

Di versi C# terbaru, ada cara yang jauh lebih ringkas untuk menulis switch jika tujuannya hanya untuk **mengembalikan nilai** ke sebuah variabel. Ini disebut **Switch Expression**.

Versi Lama:

```

1  string namaHari;
2  int hari = 1;
3  switch (hari)
4  {
5      case 1: namaHari = "Senin"; break;
6      default: namaHari = "Hari Lain"; break;
7  }
8

```

Versi Modern (Lebih Elegan):

```

1  int hari = 1;
2  string namaHari = hari switch
3  {
4      1 => "Senin",
5      2 => "Selasa",
6      3 => "Rabu",
7      _ => "Hari Lain" // '_' berfungsi sebagai default
8  };

```

Perbandingan: if-else vs switch

Fitur	if-else	switch
Kondisi	Bisa rentang (> 10), logika kompleks	Biasanya nilai tetap/konstanta
Keterbacaan	Berantakan jika cabangnya banyak	Sangat rapi untuk banyak cabang
Performa	Memeriksa satu per satu dari atas	Bisa lebih cepat (menggunakan <i>jump table</i>)

Kapan Pakai Switch?

Gunakan saat Anda membandingkan **satu variabel** terhadap daftar nilai yang sudah pasti (seperti menu, hari, status order, atau tipe user).

Gunakan jika jumlah cabangnya lebih dari 3 agar kode tidak "melebar" secara horizontal.

● 12.5 Switch Expression (C# 8.0+)

Switch Expression yang diperkenalkan pada C# 8.0 adalah evolusi dari statement switch tradisional. Jika statement switch lama fokus pada *alur eksekusi* (melakukan tindakan), Switch Expression fokus pada **menghasilkan nilai**.

Ini membuat kode Anda jauh lebih ringkas, ekspresif, dan sering kali lebih mudah dibaca.

1. Perbandingan Sintaks

Mari kita lihat perbedaannya saat ingin menentukan warna berdasarkan status kode.

Statement Switch (Lama):

```

1  string warna;
2  switch (status)
3  {
4      case 200:
5          warna = "Hijau";
6          break;
7      case 404:
8          warna = "Merah";
9          break;
10     default:
11         warna = "Abu-abu";
12         break;
13 }

```

Switch Expression (Modern):

```
1 string warna = status switch
2 {
3     200 => "Hijau",
4     404 => "Merah",
5     _ => "Abu-abu" // '_' adalah discard pattern (pengganti default)
6 };
```

2. Perubahan Aturan Penting

Ada beberapa perubahan kunci yang perlu Anda perhatikan dalam penulisan Switch Expression:

Variabel di Depan: Variabel yang dicek diletakkan sebelum kata kunci switch.

Lambda Arrow (=>): Menggantikan tanda titik dua (:) dan kata kunci case.

Tanpa break: Karena ekspresi ini langsung mengembalikan nilai, Anda tidak perlu lagi menulis break.

Discard Pattern (_): Menggantikan default. Ini wajib ada jika semua kemungkinan nilai belum terpenuhi (agar tidak terjadi error saat *runtime*).

Diakhiri Titik Koma (;): Karena ini adalah sebuah *assignment* (penugasan nilai ke variabel), seluruh blok harus diakhiri dengan ;.

3. Menggunakan Properti (Property Pattern)

Switch Expression sangat kuat saat digunakan untuk mengecek properti dari sebuah objek.

```
1 var pesan = pesan switch
2 {
3     { IsPriority: true, Status: "Pending" } => "Segera proses pengiriman prioritas!",
4     { Status: "Shipped" } => "Barang sedang dalam perjalanan.",
5     _ => "Status tidak diketahui"
6 };
```

4. Switch Expression dengan Multiple Values (Tuples)

Anda bisa membandingkan lebih dari satu variabel sekaligus tanpa perlu *nested if*.

```
1 string output = (lampu1, lampu2) switch
2 {
3     ("hijau", "hijau") => "Jalan terus",
4     ("merah", "merah") => "Berhenti semua",
5     ("merah", "hijau") => "Sisi kanan jalan",
6     _ => "Hati-hati"
7 };
8
```

5. Kapan Harus Menggunakan Switch Expression?

Gunakan Switch Expression Jika...	Gunakan Statement Switch Jika...
Anda ingin memberikan nilai ke variabel.	Anda ingin menjalankan banyak baris kode/logika di setiap case.
Logikanya sederhana (mapping A ke B).	Anda perlu memanggil banyak fungsi atau prosedur di dalam case.

Gunakan Switch Expression Jika...	Gunakan Statement Switch Jika...
Anda ingin menulis kode yang bersih (<i>clean code</i>).	Anda menggunakan versi C# di bawah 8.0.

• 12.6 Pattern Matching dalam Switch

Pattern Matching dalam switch adalah fitur yang memungkinkan Anda untuk memeriksa variabel bukan hanya berdasarkan nilainya (seperti case 1), tetapi juga berdasarkan **tipe data**, **properti**, atau **rentang nilai**.

Fitur ini membuat switch menjadi jauh lebih bertenaga (disebut *Pattern-based Switch*) dibandingkan versi tradisional.

1. Type Pattern (Mengecek Tipe Data)

Anda bisa mengecek apakah sebuah objek merupakan tipe data tertentu dan langsung memasukkannya ke dalam variabel baru.

```

1  object data = 42;
2
3  switch (data)
4  {
5      case int i:
6          Console.WriteLine($"Ini adalah angka bulat: {i}");
7          break;
8      case string s:
9          Console.WriteLine($"Ini adalah teks sepanjang {s.Length} karakter");
10         break;
11     case null:
12         Console.WriteLine("Datanya kosong (null)");
13         break;
14 }

```

2. Relational Pattern (Rentang Nilai)

(C# 9.0+) Anda bisa menggunakan operator perbandingan seperti <, >, <=, atau >= langsung di dalam case.

```

1  int nilai = 85;
2
3  string hasil = nilai switch
4  {
5      >= 90 => "A (Istimewa)",
6      >= 80 => "B (Baik)",
7      >= 70 => "C (Cukup)",
8      _ => "D (Kurang)"
9  };

```

3. Logical Pattern (and, or, not)

Anda bisa menggabungkan beberapa pola menggunakan kata kunci logika yang sangat mirip bahasa manusia.

```
1  int suhu = 25;
2
3  string status = suhu switch
4  {
5      < 0 or > 100 => "Ekstrem",
6      >= 0 and <= 30 => "Normal/Nyaman",
7      not 50 => "Bukan tepat 50 derajat",
8      _ => "Lainnya"
9  };
```

4. Property Pattern (Mengecek Properti Objek)

Anda bisa masuk ke dalam properti sebuah objek tanpa perlu melakukan *casting* manual.

```
1  var karyawan = new { Nama = "Andi", Jabatan = "Manager", MasaKerja = 5 };
2
3  string bonus = karyawan switch
4  {
5      { Jabatan: "Manager", MasaKerja: >= 5 } => "Bonus 10 Juta",
6      { Jabatan: "Staff", MasaKerja: > 10 } => "Bonus 5 Juta",
7      { Nama: "Andi" } => "Bonus Spesial Andi",
8      _ => "Tidak ada bonus"
9  };
10 var karyawan = new { Nama = "Andi", Jabatan = "Manager", MasaKerja = 5 };
11
12 string bonus = karyawan switch
13 {
14     { Jabatan: "Manager", MasaKerja: >= 5 } => "Bonus 10 Juta",
15     { Jabatan: "Staff", MasaKerja: > 10 } => "Bonus 5 Juta",
16     { Nama: "Andi" } => "Bonus Spesial Andi",
17     _ => "Tidak ada bonus"
18 };
19
```

5. Positional Pattern (Menggunakan Tuples)

Jika Anda memiliki beberapa variabel yang ingin dicek sekaligus, Anda bisa menggunakan *tuples*.

```
1  string kuadran = (x, y) switch
2  {
3      (0, 0) => "Titik Pusat",
4      (> 0, > 0) => "Kuadran 1",
5      (< 0, > 0) => "Kuadran 2",
6      _ => "Lainnya"
7  };
8
```

Keuntungan Menggunakan Pattern Matching:

Keamanan Tipe (Type Safety): Mengurangi error saat melakukan konversi tipe data (*casting*).

Keterbacaan: Menghilangkan tumpukan if-else dan pengecekan instanceof atau is.

Deklaratif: Anda fokus pada "apa yang ingin dicocokkan" daripada "bagaimana cara mengeceknya".

Bab 13: Perulangan (Looping)

- 13.1 **while** Loop: Syntax dan Penggunaan

Perulangan while adalah salah satu struktur kontrol yang paling dasar dalam C#. Berbeda dengan for loop yang biasanya digunakan saat kita sudah tahu berapa kali akan mengulang, while digunakan ketika kita ingin mengulang selama sebuah **kondisi tertentu masih terpenuhi**.

1. Sintaks Dasar

Sintaksnya sangat sederhana:

```
1  while (kondisi)
2  {
3      // Kode yang akan dijalankan berulang kali
4  }
```

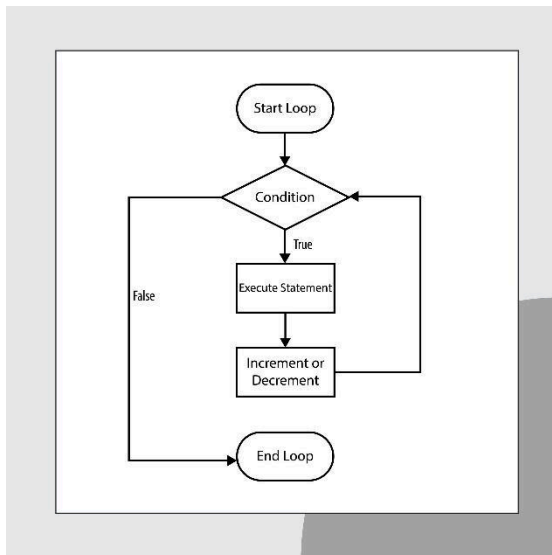
2. Contoh Penggunaan Dasar

Mari kita buat penghitung sederhana dari 1 sampai 5.

```
1  int counter = 1;
2
3  while (counter <= 5)
4  {
5      Console.WriteLine($"Iterasi ke-{counter}");
6      counter++; // Sangat penting untuk mengubah nilai kondisi
7  }
```

3. Penggunaan Nyata: Menunggu Input User

while sangat ideal untuk kasus di mana kita tidak tahu kapan perulangan akan berhenti, seperti menunggu user mengetik kata tertentu.



```

1  string input = "";
2
3  while (input != "keluar")
4  {
5      Console.Write("Ketik sesuatu (ketik 'keluar' untuk berhenti): ");
6      input = Console.ReadLine().ToLower();
7
8      if (input != "keluar")
9      {
10         Console.WriteLine($"Anda mengetik: {input}");
11     }
12 }
13
14 Console.WriteLine("Program berhenti.");
15

```

4. Perbedaan while vs do-while

Salah satu variasi dari while adalah do-while. Perbedaannya terletak pada **kapan pengecekan dilakukan**.

Jenis Loop	Pengecekan Kondisi	Jaminan Eksekusi
while	Di awal (Entry-controlled)	Mungkin 0 kali jika kondisi awal sudah false.
do-while	Di akhir (Exit-controlled)	Pasti minimal 1 kali meskipun kondisi awal false.

Contoh do-while:

```

1  int angka = 10;
2  do
3  {
4      Console.WriteLine("Ini akan muncul sekali");
5  } while (angka < 5); // Padahal 10 tidak kurang dari 5
6

```

• 13.2 do-while Loop: Perbedaan dengan while

Perbedaan mendasar antara while dan do-while terletak pada **kapan kondisi diperiksa**. Ini menentukan apakah kode di dalamnya akan dijalankan setidaknya satu kali atau tidak sama sekali.

1. Perbedaan Utama: "Cek Dulu" vs "Jalan Dulu"

- **while Loop (Entry-Controlled):** Kondisi diperiksa di **awal**. Jika kondisi sudah salah (false) sejak awal, maka kode di dalamnya **tidak akan pernah dijalankan**.
- **do-while Loop (Exit-Controlled):** Kondisi diperiksa di **akhir**. Artinya, program akan menjalankan kode di dalam blok tersebut terlebih dahulu, baru kemudian mengecek apakah harus mengulang lagi. Jadi, kode **pasti dijalankan minimal satu kali**.

2. Perbandingan Sintaks

while (Cek di Awal)	do-while (Cek di Akhir)
csharp while (kondisi) { // kode }	csharp do { // kode } while (kondisi);

Catatan Penting: Jangan lupa tanda titik koma (;) di akhir while (kondisi); pada struktur do-while. Ini adalah kesalahan sintaks yang sering dilakukan pemula.

3. Contoh Kasus Nyata

Skenario A: Kondisi yang sudah salah sejak awal

```

1  int size = 5;
2  for (int baris = 0; baris < size; baris++)
3  {
4      for (int kolom = 0; kolom < size; kolom++)
5      {
6          Console.Write("* ");
7      }
8      Console.WriteLine(); // Berpindah ke baris baru
9  }
10

```

Skenario B: Validasi Input User

do-while sangat populer digunakan untuk meminta input user, karena kita ingin user setidaknya memasukkan data satu kali sebelum kita mengecek apakah datanya valid atau apakah user ingin keluar.

```

1  string jawaban;
2  do
3  {
4      Console.WriteLine("Ketik 'ya' untuk lanjut: ");
5      jawaban = Console.ReadLine().ToLower();
6  } while (jawaban != "ya");
7
8  Console.WriteLine("Terima kasih!");
9

```

4. Kapan Menggunakan yang Mana?

- **Gunakan while jika:** Anda ingin memastikan kondisi benar-benar terpenuhi sebelum melakukan apa pun (misal: memproses data dari database yang mungkin saja kosong).
- **Gunakan do-while jika:** Anda ingin menjalankan sebuah proses setidaknya satu kali (misal: menampilkan menu permainan, meminta input password, atau melakukan percobaan koneksi pertama).

Ringkasan Cepat

Fitur	while	do-while
Pengecekan	Sebelum iterasi	Setelah iterasi
Minimal Eksekusi	0 kali	1 kali
Kegunaan Umum	Membaca file/data	Menu, Input User, Validasi

- **13.3 for Loop: Iterasi Terkontrol**

Perulangan for adalah jenis perulangan yang paling sering digunakan ketika Anda sudah mengetahui secara pasti **berapa kali** kode tersebut harus dijalankan. Berbeda dengan while yang fleksibel, for loop dirancang untuk iterasi yang sangat terkontrol.

1. Sintaks Dasar

for loop memiliki struktur yang terdiri dari tiga bagian utama di dalam satu baris:

```
C#
for (inisialisasi; kondisi; iterator)
{
    // Kode yang akan dijalankan
}
```

1. **Inisialisasi:** Membuat variabel penghitung (biasanya `int i = 0`). Dijalankan hanya satu kali di awal.
2. **Kondisi:** Syarat agar perulangan tetap berjalan (misal: `i < 5`). Dicek setiap sebelum memulai iterasi.
3. **Iterator:** Perubahan nilai penghitung (misal: `i++`). Dijalankan setiap kali blok kode selesai dieksekusi.

2. Contoh Penggunaan Dasar

Mari kita mencetak angka 1 sampai 5.

```
C#
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine("Angka: " + i);
}
```

3. Variasi Iterasi

Anda tidak selalu harus mulai dari 0 atau bertambah 1. Anda punya kontrol penuh:

- **Hitung Mundur:**

C#

```
for (int i = 10; i > 0; i--) { ... }
```

- **Lompat Angka (Ganjil/Genap):**

C#

```
for (int i = 0; i <= 10; i += 2) { ... } // Naik per 2 angka
```

4. Menggunakan for dengan Array

for loop sangat cocok untuk memproses elemen dalam array karena kita bisa menggunakan variabel i sebagai indeks.

C#

```
string[] buah = { "Apel", "Pisang", "Jeruk" };
```

```
for (int i = 0; i < buah.Length; i++)
```

```
{
```

```
    Console.WriteLine($"Indeks ke-{i} adalah {buah[i]}");
```

```
}
```

5. Nested for Loop (Perulangan Bersarang)

Anda bisa meletakkan for di dalam for. Ini sering digunakan untuk mengolah data berbentuk tabel (baris dan kolom) atau koordinat.

C#

```
for (int baris = 1; baris <= 3; baris++)
```

```
{
```

```
    for (int kolom = 1; kolom <= 3; kolom++)
```

```
    {
```

```
        Console.Write($"({baris},{kolom}) ");
```

```
    }
```

```
    Console.WriteLine(); // Baris baru setelah kolom selesai
```

```
}
```

Perbandingan: for vs while

Fitur	for Loop	while Loop
Kapan digunakan?	Sudah tahu jumlah iterasinya.	Jumlah iterasi tidak pasti (dinamis).
Keterbacaan	Sangat rapi (semua kontrol di satu baris).	Lebih menyebar (variabel dibuat di luar).
Risiko Infinite Loop	Rendah (karena iterator ada di header).	Lebih tinggi (sering lupa update variabel).

Tips Pro: Menghindari "Off-by-One Error"

Kesalahan paling umum pada for loop adalah menentukan apakah harus menggunakan < atau <=.

- Jika berurusan dengan **Array/List**, gunakan selalu < namaArray.Length karena indeks dimulai dari 0.
- Jika menghitung **jumlah objek** secara alami (1 sampai 10), gunakan <=.

- **13.4 foreach Loop: Iterasi Koleksi**

foreach adalah cara yang paling elegan dan aman untuk membaca data dari sebuah koleksi (seperti Array, List, atau Dictionary) di C#. Berbeda dengan for loop tradisional, Anda tidak perlu berurusan dengan indeks ([i]) atau menghitung panjang koleksi secara manual.

1. Sintaks Dasar

Sintaks foreach dirancang agar terbaca seperti kalimat bahasa Inggris: *"Untuk setiap item di dalam koleksi, lakukan sesuatu."*

```
C#
foreach (tipeData namaVariabel in namaKoleksi)
{
    // Kode yang dijalankan untuk setiap item
}
```

Contoh Sederhana:

```
C#
string[] daftarBuah = { "Apel", "Mangga", "Jeruk" };

foreach (string buah in daftarBuah)
{
    Console.WriteLine($"Saya suka makan {buah}");
}
```

2. Keunggulan Utama foreach

- **Readability (Keterbacaan):** Kode jauh lebih bersih dan ringkas. Tidak ada int i = 0 atau i++ yang memenuhi layar.
- **Keamanan (Safety):** Menghilangkan risiko **IndexOutOfRangeException** (error karena memanggil indeks yang tidak ada).
- **Abstraksi:** Bisa digunakan pada semua jenis koleksi yang mendukung IEnumerable, baik itu Array sederhana maupun struktur data kompleks seperti Dictionary.

3. Menggunakan var agar Lebih Fleksibel

Dalam pengembangan modern, programmer sering menggunakan kata kunci var agar C# secara otomatis menentukan tipe datanya. Ini sangat membantu jika tipe datanya panjang atau kompleks.

```
C#
var angka = new List<int> { 10, 20, 30, 40 };
```

```
foreach (var n in angka)
```

```
{
    Console.WriteLine(n);
}
```

4. Batasan Penting: Read-Only

Ada satu aturan emas dalam foreach: **Anda tidak boleh mengubah isi koleksi (menambah atau menghapus elemen) saat perulangan sedang berjalan.**

- **Boleh:** Mengubah properti dari objek di dalam koleksi.
- **Tidak Boleh:** Mencoba menghapus item dari list di tengah-tengah foreach. Jika dilakukan, program akan melempar *exception* (error).
- **Tidak Boleh:** Mencoba mengganti nilai variabel iterasi (misalnya: $n = 5$ di dalam loop). Variabel iterasi bersifat *read-only*.

5. Kapan Pakai foreach vs for?

Skenario	Gunakan foreach	Gunakan for
Membaca semua data	✓ (Pilihan Utama)	✗ (Terlalu bertele-tele)
Butuh informasi indeks	✗ (Tidak punya indeks)	✓ (Bisa pakai i)
Mengubah isi koleksi	✗ (Akan error)	✓ (Aman dilakukan)
Iterasi Terbalik (Mundur)	✗ (Hanya bisa maju)	✓ (Bisa pakai i--)
Lompat-lompat item	✗ (Urut satu per satu)	✓ (Bisa pakai i += 2)

Tips Wit:

Jika Anda hanya ingin menampilkan isi list ke layar atau menjumlahkan angka, foreach adalah sahabat terbaik Anda. Namun, jika Anda butuh tahu "ini item

keberapa ya?", Anda bisa menggunakan for loop atau menggunakan trik index manual di luar foreach.

- **13.5 break dan continue: Mengontrol Alur Loop**

Dalam perulangan, terkadang kita butuh "rem darurat" atau "tombol skip" untuk mengontrol alur tanpa harus menunggu kondisi utama perulangan selesai. Di sinilah break dan continue berperan.

1. break: Keluar Secara Paksa

break digunakan untuk menghentikan seluruh proses perulangan seketika itu juga. Program akan langsung melompat ke baris kode setelah blok perulangan.

- **Gunakan saat:** Anda sudah menemukan data yang dicari atau terjadi kesalahan yang mengharuskan loop berhenti.

Contoh:

```
C#
for (int i = 1; i <= 10; i++)
{
    if (i == 5)
    {
        break; // Berhenti total saat i mencapai 5
    }
    Console.WriteLine(i);
}
// Output: 1, 2, 3, 4
```

2. continue: Lewati Iterasi Ini

continue tidak menghentikan seluruh perulangan, melainkan hanya **melewati (skip)** sisa kode di dalam iterasi saat ini dan langsung lanjut ke iterasi berikutnya.

- **Gunakan saat:** Anda ingin mengabaikan item tertentu (seperti data kosong atau angka genap) tapi tetap ingin melanjutkan sisa perulangan.

Contoh:

```
C#
for (int i = 1; i <= 5; i++)
{
    if (i == 3)
    {
        continue; // Lewati angka 3, tapi lanjut ke angka 4
    }
    Console.WriteLine(i);
}
// Output: 1, 2, 4, 5
```

3. Perbedaan Visual

Fitur	break	continue
Efek	Mengakhiri seluruh perulangan.	Melewati iterasi saat ini saja.
Langkah Selanjutnya	Keluar dari blok for/while.	Kembali ke atas (cek kondisi/increment).
Analogi	Menutup toko karena barang sudah habis.	Melewati pelanggan yang tidak bawa uang.

4. Contoh Kasus Nyata: Mencari Data

Misalkan kita mencari angka tertentu dalam sebuah array. Jika sudah ketemu, tidak perlu lagi mengecek sisanya (efisiensi memori).

```
1  int[] angka = { 10, 20, 30, 40, 50 };
2  int cari = 30;
3
4  foreach (int n in angka)
5  {
6      if (n == cari)
7      {
8          Console.WriteLine("Data ditemukan!");
9          break; // Tidak perlu cek 40 dan 50
10     }
11     Console.WriteLine($"Memeriksa {n}...");
12 }
```

● 13.6 Infinite Loop dan Cara Menghindarinya

Infinite Loop (perulangan tak terbatas) adalah situasi di mana kode perulangan terus berjalan selamanya karena kondisi berhentinya tidak pernah tercapai. Hal ini dapat menyebabkan aplikasi membeku (*hang*), penggunaan CPU mencapai 100%, hingga sistem mengalami *crash*.

1. Penyebab Umum Infinite Loop

A. Lupa Memperbarui Variabel (Iterator)

Ini adalah penyebab paling klasik. Variabel yang seharusnya berubah agar kondisi menjadi false justru tidak pernah diubah.

C#

```
int i = 0;
while (i < 5)
{
```

```

    Console.WriteLine("Halo!");
    // Lupa menambahkan i++;
}

```

B. Kondisi yang Selalu Benar

Terkadang logika yang kita tulis secara matematis tidak akan pernah bernilai false.

C#

```

for (int i = 10; i > 0; i++) // i bertambah terus, sehingga i akan selalu > 0
{
    // Kode ini tidak akan pernah berhenti
}

```

C. Kesalahan Perbandingan pada Tipe Floating-Point

Menggunakan double atau float dalam kondisi perulangan sangat berisiko karena masalah presisi desimal.

C#

```

double d = 0;
while (d != 1.0) // Karena masalah presisi, d mungkin menjadi 0.9999999999 dan melompati 1.0
{
    d += 0.1;
}

```

2. Cara Menghindari Infinite Loop

Berikut adalah strategi praktis agar kode Anda tetap aman:

1. Gunakan for Loop Jika Memungkinkan

for loop memaksa Anda menulis inisialisasi, kondisi, dan iterator dalam satu baris. Ini meminimalkan risiko lupa memperbarui variabel.

2. Gunakan "Safety Counter" (Batas Maksimal)

Jika Anda menggunakan while untuk proses yang tidak pasti (seperti memanggil API), tambahkan batas maksimal percobaan.

C#

```

int attempts = 0;
bool success = false;

while (!success && attempts < 100)
{
    success = TryConnect();
    attempts++;
}

```

3. Periksa Logika Arah (Increment/Decrement)

Pastikan jika kondisi Anda menggunakan "kurang dari" (<), variabelnya harus bertambah (++). Jika menggunakan "lebih dari" (>), variabelnya harus berkurang (--).

4. Gunakan break sebagai Pintu Keluar Darurat

Jangan ragu menggunakan break jika ada kondisi aneh yang terdeteksi di dalam loop.

C#

```

while (true) // Loop sengaja dibuat true
{
    string input = Console.ReadLine();
}

```

```
    if (string.IsNullOrEmpty(input) || input == "exit") break;
}
```

3. Cara Menghentikan Infinite Loop Saat Running

Jika Anda terlanjur menjalankan program yang terjebak dalam infinite loop:

Di Visual Studio: Tekan tombol **Stop** (ikon kotak merah) atau tekan Shift + F5.

Di Terminal/Console: Tekan Ctrl + C untuk memaksa proses berhenti.

- **13.7 Nested Loops dan Contoh Aplikasi**

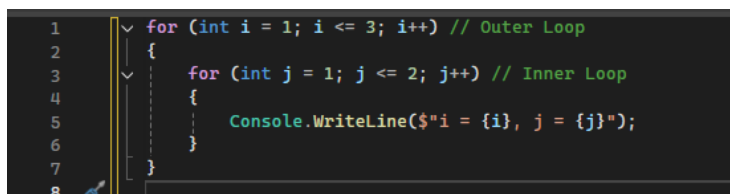
Nested Loop adalah struktur di mana sebuah perulangan berada di dalam perulangan lainnya. Dalam konsep ini, setiap satu kali iterasi pada **Outer Loop** (perulangan luar), **Inner Loop** (perulangan dalam) akan dijalankan sepenuhnya hingga selesai.

1. Cara Kerja Nested Loop Bayangkan Anda sedang melihat jam dinding. Jarum jam hanya bergeser satu angka setelah jarum menit berputar penuh sebanyak 60 kali.

Outer Loop: Jarum Jam.

Inner Loop: Jarum Menit.

Struktur Dasar:

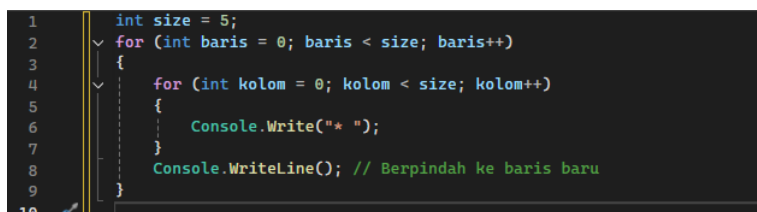


```
1  for (int i = 1; i <= 3; i++) // Outer Loop
2  {
3      for (int j = 1; j <= 2; j++) // Inner Loop
4      {
5          Console.WriteLine($"i = {i}, j = {j}");
6      }
7  }
```

2. Contoh Aplikasi Nyata

A. Membuat Pola Grafis (Matriks/Grid)

Ini adalah latihan paling umum untuk memahami koordinat. Misalnya, membuat persegi bintang:



```
1  int size = 5;
2  for (int baris = 0; baris < size; baris++)
3  {
4      for (int kolom = 0; kolom < size; kolom++)
5      {
6          Console.Write("* ");
7      }
8      Console.WriteLine(); // Berpindah ke baris baru
9  }
```

B. Mengolah Data Tabel (Array 2 Dimensi)

Jika Anda memiliki data Excel atau tabel database, Anda menggunakan nested loop untuk membaca baris dan kolomnya.

```
1  int[,] matriks = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
2
3  for (int i = 0; i < 3; i++) // Baris
4  {
5      for (int j = 0; j < 2; j++) // Kolom
6      {
7          Console.Write(matriks[i, j] + " ");
8      }
9      Console.WriteLine();
10 }
```

C. Kombinasi Produk atau Kemungkinan

Misalnya, Anda ingin menampilkan semua kombinasi warna baju dan ukuran yang tersedia.

```
1  string[] warna = { "Merah", "Biru" };
2  string[] ukuran = { "S", "M", "L" };
3
4  foreach (var w in warna)
5  {
6      foreach (var u in ukuran)
7      {
8          Console.WriteLine($"{w} ukuran {u}");
9      }
10 }
```

Bab 14: Array dan Koleksi Dasar

• 14.1 Deklarasi dan Inisialisasi Array

Array adalah struktur data yang memungkinkan Anda menyimpan sekumpulan nilai dengan tipe data yang sama dalam satu variabel. Bayangkan array seperti sebuah rak buku dengan nomor indeks yang tetap.

1. Sintaks Dasar Deklarasi

Untuk mendeklarasikan array, Anda cukup menambahkan kurung siku [] setelah tipe datanya.

C#

// Deklarasi saja (belum ada memorinya)

string[] daftarNama;

int[] kumpulanAngka;

2. Deklarasi dengan Inisialisasi (Alokasi Memori)

Agar array bisa digunakan, Anda harus menentukan ukurannya menggunakan kata kunci new.

C#

```
// Membuat array kosong dengan kapasitas 5 elemen
```

```
int[] angka = new int[5];
```

```
// Default value:
```

```
// int akan berisi 0, bool berisi false, string berisi null.
```

3. Inisialisasi dengan Nilai Awal

Ada beberapa cara singkat untuk langsung mengisi data ke dalam array saat dideklarasikan:

A. Cara Lengkap

C#

```
string[] hobi = new string[3] { "Coding", "Gaming", "Reading" };
```

B. Cara Singkat (Paling Sering Digunakan)

C# cukup pintar untuk menghitung jumlah elemen secara otomatis.

C#

```
int[] skor = { 90, 85, 70, 100 };
```

```
string[] mobil = { "Toyota", "Honda", "Tesla" };
```

4. Karakteristik Penting Array

1. **Indeks Dimulai dari 0:** Elemen pertama selalu berada di indeks 0, elemen kedua di indeks 1, dan seterusnya.
2. **Properti Length:** Digunakan untuk mengetahui jumlah total elemen.
3. **Akses Elemen:** Menggunakan kurung siku [].

Contoh Akses Data:

```
1  string[] planet = { "Merkurius", "Venus", "Bumi" };
2
3  Console.WriteLine(planet[0]); // Output: Merkurius
4  planet[2] = "Mars";           // Mengubah "Bumi" menjadi "Mars"
5  Console.WriteLine(planet.Length); // Output: 3
6
```

5. Array Multidimensi (Matriks)

C# juga mendukung array yang memiliki baris dan kolom.

```
1 // Array 2D (3 Baris, 2 Kolom)
2 int[,] koordinat = new int[3, 2] {
3     { 1, 2 },
4     { 3, 4 },
5     { 5, 6 }
6 };
7
8 Console.WriteLine(koordinat[0, 1]); // Output: 2
```

● 14.2 Mengakses dan Memodifikasi Elemen Array

Setelah kita mendeklarasikan dan memberikan nilai pada array, langkah selanjutnya adalah berinteraksi dengan datanya. Di C#, akses dan modifikasi elemen dilakukan menggunakan **Indeks**.

1. Mengakses Elemen Array

Setiap kotak di dalam array memiliki nomor urut yang disebut **indeks**. Ingat aturan emasnya: **Indeks selalu dimulai dari 0**.

Elemen ke-1 = Indeks 0

Elemen ke-2 = Indeks 1

Elemen ke-n = Indeks n - 1

```
1 string[] superHero = { "Iron Man", "Thor", "Hulk", "Captain America" };
2
3 // Mengambil nilai "Iron Man"
4 string heroPertama = superHero[0];
5
6 // Mengambil nilai "Hulk"
7 Console.WriteLine(superHero[2]);
8
```

2. Memodifikasi Elemen Array

Array bersifat *mutable* (isinya bisa diubah), meskipun ukurannya tetap. Untuk mengubah nilai, Anda cukup menunjuk indeksnya dan memberikan nilai baru menggunakan operator penugasan (=).

```
1 string[] buah = { "Apel", "Jeruk", "Pisang" };
2
3 // Mengubah "Jeruk" menjadi "Mangga"
4 buah[1] = "Mangga";
5
6 Console.WriteLine(buah[1]); // Sekarang hasilnya: Mangga
7
```

3. Mengakses Elemen dari Belakang (C# 8.0+)

C# modern memperkenalkan operator "Hat" (^) yang memudahkan kita mengambil data dari urutan paling akhir tanpa harus menghitung Length - 1.

[^1] = Elemen terakhir.

[^2] = Elemen kedua dari belakang.

```
1 int[] angka = { 10, 20, 30, 40, 50 };
2
3 int terakhir = angka[^1]; // Hasil: 50
4 int duaTerakhir = angka[^2]; // Hasil: 40
5
```

4. Kesalahan Paling Umum: IndexOutOfRangeException

Error ini terjadi jika Anda mencoba mengakses indeks yang tidak ada di dalam jangkauan array.

```
11 0454[3] = 100; // ERROR! Karena indeks 3 adalah elemen ke-4, sedangkan kapasitas hanya 3
12
13
14 int[] array = new int[3]; // Indeks yang tersedia: 0, 1, 2
```

5. Properti Penting: Length

Selalu gunakan properti .Length untuk mengetahui jumlah total elemen agar Anda tidak salah menebak indeks maksimal.

```
12
11 int elemenTerakhir = kumpulanAngka[kumpulanAngka.Length - 1];
13 // Mendapatkan elemen terakhir secara otomatis
14
15 int[] kumpulanAngka = { 10, 80, 20, 82 };
```

• 14.3 Looping Melalui Array dengan **for** dan **foreach**

Setelah Anda tahu cara mengakses satu elemen, sekarang saatnya belajar cara memproses **seluruh** isi array secara otomatis. Dalam C#, ada dua cara utama yang masing-masing punya "kekuatan" berbeda.

1. Menggunakan for Loop (Kontrol Penuh)

Gunakan for jika Anda perlu memodifikasi isi array atau membutuhkan informasi **indeks** (urutan ke berapa).

C#

```
string[] daftarTugas = { "Belajar C#", "Beli Kopi", "Olahraga" };
```

```
for (int i = 0; i < daftarTugas.Length; i++)
{
    // Mengubah data: Menambahkan nomor urut di depan teks
    daftarTugas[i] = $"{i + 1}. {daftarTugas[i]}";
    Console.WriteLine(daftarTugas[i]);
}
```

- **Kelebihan:** Bisa akses indeks, bisa iterasi mundur (i--), dan bisa mengubah nilai elemen secara langsung.
- **Kekurangan:** Penulisan lebih panjang dan ada risiko *Index Out of Range* jika salah menulis batas kondisi.

2. Menggunakan foreach Loop (Ringkas & Aman)

Gunakan foreach jika Anda hanya ingin **membaca** atau menampilkan semua data satu per satu tanpa peduli indeksinya.

C#

```
int[] kumpulanSkor = { 85, 92, 78, 90 };
int total = 0;
```

```
foreach (int skor in kumpulanSkor)
{
    total += skor; // Menghitung total nilai
}
```

Console.WriteLine(\$"Total Skor: {total}");

- **Kelebihan:** Sangat mudah dibaca, lebih aman (tidak mungkin salah indeks), dan kodenya bersih.
- **Kekurangan:** Tidak bisa mengubah isi elemen array (bersifat *read-only*) dan tidak tahu data tersebut ada di urutan ke berapa secara otomatis.

Perbandingan Penggunaan

Fitur	for Loop	foreach Loop
Tujuan Utama	Manipulasi data & akses indeks.	Membaca/Iterasi data saja.
Modifikasi Elemen	Bisa (misal: <code>arr[i] = 10</code>)	Tidak Bisa (error jika mencoba mengubah)
Arah Iterasi	Bisa Maju atau Mundur.	Selalu Maju (dari awal ke akhir).
Keterbacaan	Sedikit lebih kompleks.	Sangat "manusiawi" (elegan).

Contoh: Cari Angka Tertinggi

Berikut cara mencari nilai maksimum dalam array menggunakan foreach:

```
1 int[] angka = { 4, 12, 7, 20, 10 };
2 int max = angka[0];
3
4 foreach (int n in angka)
5 {
6     if (n > max) max = n;
7 }
8
9 Console.WriteLine($"Nilai tertinggi adalah: {max}");
10
```

- **14.4 Metode Array:** `Sort()`, `Reverse()`, `IndexOf()`

C# menyediakan *class* Array yang berisi berbagai metode statis untuk memanipulasi data tanpa kita harus menulis logika manual dari nol. Tiga metode yang paling sering digunakan adalah `Sort()`, `Reverse()`, dan `IndexOf()`.

Berikut adalah panduan cara menggunakannya:

1. `Array.Sort()`: Mengurutkan Data

Metode ini digunakan untuk mengurutkan elemen array secara *ascending* (dari kecil ke besar untuk angka, atau A ke Z untuk string).

- **Sifat:** Mengubah array asli secara langsung.
- **Penggunaan:**

C#

```
int[] angka = { 5, 2, 8, 1, 9 };
```

```
Array.Sort(angka);
```

```
// Hasil: { 1, 2, 5, 8, 9 }  
Console.WriteLine(string.Join(", ", angka));
```

2. Array.Reverse(): Membalik Urutan

Metode ini membalikkan posisi elemen dalam array. Elemen pertama menjadi terakhir, dan elemen terakhir menjadi pertama.

- **Tips:** Jika ingin mengurutkan dari besar ke kecil (*descending*), lakukan Sort() terlebih dahulu, baru kemudian Reverse().

- **Penggunaan:**

```
C#  
string[] huruf = { "A", "B", "C" };  
Array.Reverse(huruf);
```

```
// Hasil: { "C", "B", "A" }
```

3. Array.IndexOf(): Mencari Posisi Elemen

Metode ini digunakan untuk mencari **indeks** dari nilai tertentu di dalam array.

- **Hasil:** Jika ketemu, mengembalikan nomor indeksnya (0, 1, 2...). Jika **tidak ketemu**, mengembalikan nilai -1.

- **Penggunaan:**

```
C#  
string[] buah = { "Apel", "Mangga", "Jeruk" };  
int posisi = Array.IndexOf(buah, "Mangga");  
  
if (posisi != -1) {  
    Console.WriteLine($"Mangga ada di indeks ke-{posisi}");  
} else {  
    Console.WriteLine("Buah tidak ditemukan.");  
}
```

Ringkasan Perbandingan

Metode	Input Utama	Hasil/Output	Mengubah Array Asli?
Sort()	Array	Array terurut (A-Z / 0-9)	Ya
Reverse()	Array	Array dengan urutan terbalik	Ya
IndexOf()	Array, Nilai Cari	int (indeks atau -1)	Tidak

Contoh Kombinasi: Mengurutkan Terbalik (Descending)

Karena C# tidak memiliki Array.SortDescending(), kita menggabungkan dua metode:

```
C#  
int[] nilai = { 40, 100, 10, 50 };  
  
Array.Sort(nilai); // Jadi: 10, 40, 50, 100
```

```
Array.Reverse(nilai); // Jadi: 100, 50, 40, 10
```

```
Console.WriteLine(nilai[0]); // Output: 100
```

- **14.5 Array Multidimensi dan Jagged Array**

Dalam pemrograman (misalnya di C#), **Array Multidimensi** dan **Jagged Array** sama-sama digunakan untuk menyimpan data dalam bentuk lebih dari satu dimensi, tetapi strukturnya berbeda.

- ◆ **1. Array Multidimensi**

Array multidimensi adalah array yang memiliki jumlah baris dan kolom **tetap (simetris)**.

Ciri-ciri:

- Bentuknya seperti tabel (matrix)
- Jumlah kolom di setiap baris sama
- Ditulis dengan tanda koma [,]

Contoh:

```
int[,] angka = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

Artinya:

- 2 baris
- 3 kolom

Mengakses elemen:

```
Console.WriteLine(angka[0,1]); // Output: 2
```

- ◆ **2. Jagged Array**

Jagged array adalah array yang berisi array lain, tetapi **panjang tiap baris bisa berbeda (tidak simetris)**.

Ciri-ciri:

- Disebut juga "array bertingkat"
- Setiap baris bisa memiliki jumlah kolom berbeda
- Ditulis dengan [][]

Contoh:

```
int[][] angka = new int[2][];
```

```
angka[0] = new int[] {1, 2, 3};
```

```
angka[1] = new int[] {4, 5};
```

Artinya:

- Baris pertama punya 3 elemen
- Baris kedua hanya 2 elemen

Mengakses elemen:

```
Console.WriteLine(angka[1][0]); // Output: 4
```

- ◆ **Perbedaan Utama**

Aspek	Array Multidimensi	Jagged Array
Struktur	Rapi (matrix)	Tidak beraturan
Panjang kolom	Sama semua	Bisa berbeda
Penulisan	[.]	[]
Fleksibilitas	Kurang fleksibel	Lebih fleksibel

● 14.6 Pengenalan `List<T>` dan Koleksi Generik

◆ 1. Apa itu Koleksi Generik?

Koleksi Generik (Generic Collection) adalah struktur data yang bisa menyimpan banyak data **dengan tipe yang sama secara aman**.

Kelebihan:

- Lebih **aman (type-safe)** → tidak perlu casting
- Lebih **fleksibel**
- Performa lebih baik dibanding koleksi non-generik

Contoh koleksi generik:

- List<T>
 - Dictionary<TKey, TValue>
 - Queue<T>
 - Stack<T>
-

◆ 2. Apa itu List<T>?

List<T> adalah koleksi generik yang paling sering digunakan, mirip seperti array tapi **ukurannya bisa berubah (dinamis)**.

T artinya **tipe data** (misalnya int, string, dll)

◆ 3. Cara Membuat List

```
using System.Collections.Generic;
```

```
List<int> angka = new List<int>();
```

Atau langsung isi:

```
List<string> nama = new List<string>() { "Ani", "Budi", "Cici" };
```

♦ 4. Operasi Dasar pada List

Menambah data

```
angka.Add(10);
```

```
angka.Add(20);
```

Menampilkan data

```
foreach (int item in angka)
```

```
{  
    Console.WriteLine(item);  
}
```

Mengakses elemen

```
Console.WriteLine(angka[0]); // elemen pertama
```

Menghapus data

```
angka.Remove(10); // hapus berdasarkan nilai
```

```
angka.RemoveAt(0); // hapus berdasarkan index
```

Mengetahui jumlah data

```
Console.WriteLine(angka.Count);
```

♦ 5. Kelebihan List<T> dibanding Array

Array	List
Ukuran tetap	Ukuran dinamis
Kurang fleksibel	Lebih fleksibel
Manual resize	Otomatis

♦ 6. Contoh Program Sederhana

```

1  using System;
2  using System.Collections.Generic;
3
4  class Program
5  {
6      static void Main()
7      {
8          List<string> buah = new List<string>();
9
10         buah.Add("Apel");
11         buah.Add("Mangga");
12         buah.Add("Jeruk");
13
14         foreach (string item in buah)
15         {
16             Console.WriteLine(item);
17         }
18     }
19 }

```

BAGIAN 3: METODE DAN PEMROGRAMAN TERSTRUKTUR

Bab 15: Metode (Fungsi) dalam C#

- 15.1 Definisi dan Pemanggilan Metode

- 1. Apa itu Metode?

Metode (Method) adalah sekumpulan kode yang digunakan untuk melakukan suatu tugas tertentu.

Metode membantu agar program:

- Lebih rapi
- Mudah dibaca
- Bisa digunakan ulang

- 2. Struktur (Sintaks) Metode

tipeData NamaMetode(parameter)

```

{
    // isi kode
}

```

Penjelasan:

- tipeData → tipe nilai yang dikembalikan (contoh: int, string, void)

- NamaMetode → nama fungsi
 - parameter → input (boleh kosong)
 - {} → isi metode
-

♦ 3. Contoh Definisi Metode

Metode tanpa nilai balik

```
void Sapa()  
  
{  
    Console.WriteLine("Halo!");  
}
```

Metode dengan parameter

```
void SapaNama(string nama)  
  
{  
    Console.WriteLine("Halo " + nama);  
}
```

Metode dengan nilai balik

```
int Tambah(int a, int b)  
  
{  
    return a + b;  
}
```

♦ 4. Pemanggilan Metode

Metode dipanggil di dalam Main() atau metode lain.

Contoh:

```
static void Main()  
  
{  
    Sapa(); // memanggil metode
```

```
SapaNama("Budi");  
  
int hasil = Tambah(5, 3);  
  
Console.WriteLine(hasil);  
  
}
```

♦ 5. Jenis Metode

1. Tanpa parameter & tanpa return

```
void Halo()  
  
{  
  
    Console.WriteLine("Hai!");  
  
}
```

2. Dengan parameter, tanpa return

```
void Cetak(int angka)  
  
{  
  
    Console.WriteLine(angka);  
  
}
```

3. Tanpa parameter, dengan return

```
int AmbilAngka()  
  
{  
  
    return 10;  
  
}
```

4. Dengan parameter & return

```
int Kali(int a, int b)  
  
{  
  
    return a * b;  
  
}
```

◆ 6. Contoh Program Lengkap

```
1 using System;
2
3 class Program
4 {
5     1 reference
6     static void Sapa(string nama)
7     {
8         Console.WriteLine("Halo " + nama);
9     }
10
11     1 reference
12     static int Tambah(int x, int y)
13     {
14         return x + y;
15     }
16
17     0 references
18     static void Main()
19     {
20         Sapa("Ani");
21         int hasil = Tambah(4, 6);
22         Console.WriteLine("Hasil: " + hasil);
23     }
24 }
```

● 15.2 Parameter Metode: Value vs Reference (**ref**, **out**)

◆ 1. Parameter Value (By Value)

Parameter **value** berarti nilai variabel **disalin** ke metode.

Perubahan di dalam metode **tidak mempengaruhi** variabel asli.

Contoh:

```
void Ubah(int x)
{
    x = 10;
}
```

```
int a = 5;
```

```
Ubah(a);
```

```
Console.WriteLine(a); // Output: 5 (tidak berubah)
```

Karena yang dikirim hanya **salinan nilai**, bukan variabel asli.

◆ 2. Parameter Reference (By Reference)

Parameter **reference** berarti yang dikirim adalah **alamat variabel asli**.

Perubahan di dalam metode **akan mempengaruhi** variabel asli.

◆ 3. Menggunakan ref

Ciri-ciri:

-Harus **diinisialisasi dulu**

-Menggunakan keyword **ref** saat deklarasi dan pemanggilan

Contoh:

```
void Ubah(ref int x)
{
    x = 10;
}
```

```
int a = 5;
```

Ubah(ref a);

Console.WriteLine(a); // Output: 10

♦ 4. Menggunakan out

Ciri-ciri:

- Tidak perlu nilai awal
- Wajib diisi di dalam metode
- Menggunakan keyword out

Contoh:

```
void AmbilNilai(out int x)
{
    x = 20;
}
```

int a;

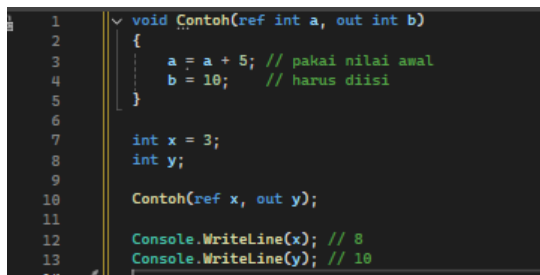
AmbilNilai(out a);

Console.WriteLine(a); // Output: 20

♦ Perbedaan ref vs out

Aspek	ref	out
Nilai awal	Harus ada	Tidak perlu
Harus diisi	Tidak wajib	Wajib diisi dalam method
Tujuan umum	Mengubah nilai	Mengembalikan nilai

♦ 5. Contoh Perbandingan



```
1 void Contoh(ref int a, out int b)
2 {
3     a = a + 5; // pakai nilai awal
4     b = 10;   // harus diisi
5 }
6
7 int x = 3;
8 int y;
9
10 Contoh(ref x, out y);
11
12 Console.WriteLine(x); // 8
13 Console.WriteLine(y); // 10
14
```

● 15.3 Default Parameter dan Named Arguments

♦ 1. Default Parameter

Default parameter adalah parameter yang memiliki **nilai bawaan (default)**, jadi saat pemanggilan metode, parameter itu boleh tidak diisi.

Tujuan:

- Mempermudah pemanggilan metode
- Mengurangi jumlah parameter yang wajib diisi

Contoh:

```
void Sapa(string nama = "Teman")
{
    Console.WriteLine("Halo " + nama);
}
```

```
}
```

Pemanggilan:

```
Sapa();           // Output: Halo Teman  
Sapa("Budi");    // Output: Halo Budi
```

♦ **2. Aturan Default Parameter**

- Harus diletakkan **di akhir parameter**
- Nilainya harus **konstan (tetap)**

Salah:

```
void Contoh(int a = 1, int b) // ERROR
```

Benar:

```
void Contoh(int b, int a = 1)
```

♦ **3. Named Arguments**

Named arguments adalah cara memanggil metode dengan menyebutkan **nama parameter secara langsung**.

Tidak harusurut sesuai posisi

Contoh:

```
void Data(string nama, int umur)  
{  
    Console.WriteLine(nama + " berumur " + umur);  
}
```

Pemanggilan:

```
Data(nama: "Ani", umur: 17);  
Data(umur: 17, nama: "Ani"); // tetap benar walau urutan dibalik
```

♦ **4. Kombinasi Default + Named Arguments**

```
void Info(string nama = "Anonim", int umur = 0)  
{  
    Console.WriteLine(nama + " - " + umur);  
}
```

Pemanggilan:

```
Info();           // Anonim - 0  
Info("Budi");    // Budi - 0  
Info(umur: 18);   // Anonim - 18  
Info(nama: "Siti", umur: 20); // Siti - 20
```

● **15.4 Return Values dan Multiple Returns dengan Tuple**

Berikut penjelasan **Return Values dan Multiple Returns** dalam C# secara jelas 🙌

♦ **1. Return Value (Nilai Kembalian)**

Return value adalah nilai yang dikembalikan oleh suatu metode setelah dijalankan. Digunakan saat metode menghasilkan suatu hasil.

Contoh:

```
int Tambah(int a, int b)  
{
```

```
    return a + b;
}
```

Pemanggilan:

```
int hasil = Tambah(5, 3);
Console.WriteLine(hasil); // Output: 8
```

♦ **2. Fungsi void**

Jika metode tidak mengembalikan nilai, gunakan void.

```
void Sapa()
{
    Console.WriteLine("Halo!");
}
```

♦ **3. Multiple Returns (Lebih dari Satu Nilai)**

Di C#, metode hanya bisa punya **1 return**, tapi bisa mengembalikan **lebih dari satu nilai** dengan beberapa cara:

♦ **Cara 1: Menggunakan out**

```
void Hitung(int a, int b, out int jumlah, out int kali)
{
    jumlah = a + b;
    kali = a * b;
}
```

Pemanggilan:

```
int j, k;
Hitung(4, 5, out j, out k);
```

```
Console.WriteLine(j); // 9
Console.WriteLine(k); // 20
```

♦ **Cara 2: Menggunakan Tuple (Paling modern)**

```
(int, int) Hitung(int a, int b)
{
    return (a + b, a * b);
}
```

Pemanggilan:

```
var hasil = Hitung(4, 5);
```

```
Console.WriteLine(hasil.Item1); // 9
Console.WriteLine(hasil.Item2); // 20
```

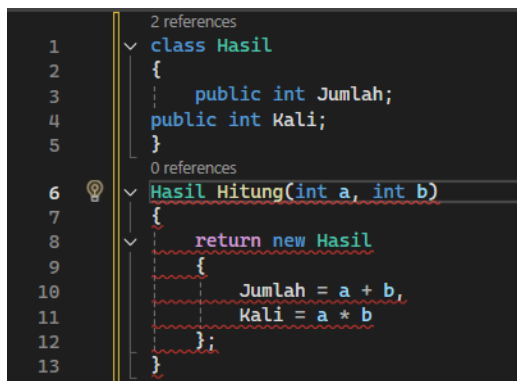
Atau pakai nama biar lebih jelas:

```
(int jumlah, int kali) Hitung(int a, int b)
{
    return (a + b, a * b);
}
```

```
var data = Hitung(4, 5);
```

```
Console.WriteLine(data.jumlah);
Console.WriteLine(data.kali);
```

♦ Cara 3: Menggunakan Class / Object



```
1  class Hasil
2  {
3      public int Jumlah;
4      public int Kali;
5  }
6
7  Hasil Hitung(int a, int b)
8  {
9      return new Hasil
10     {
11         Jumlah = a + b,
12         Kali = a * b
13     };
14 }
```

● 15.5 Void Methods vs Value-Returning Methods

♦ 1. Void Methods

Void method adalah metode yang **tidak mengembalikan nilai**.

Biasanya digunakan untuk:

- Menampilkan output
- Menjalankan aksi/proses tertentu

Contoh:

```
void Sapa()
{
    Console.WriteLine("Halo!");
}
```

Pemanggilan:

```
Sapa(); // hanya menjalankan, tidak menghasilkan nilai
```

♦ 2. Value-Returning Methods

Value-returning method adalah metode yang **mengembalikan nilai** menggunakan return.

Digunakan saat kita butuh hasil dari proses.

Contoh:

```
int Tambah(int a, int b)
{
    return a + b;
}
```

Pemanggilan:

```
int hasil = Tambah(5, 3);
Console.WriteLine(hasil); // Output: 8
```

♦ Perbedaan Utama

Aspek	Void Method	Value-Returning Method
Nilai balik	Tidak ada	Ada (return)
Tipe data	void	int, string, dll

Aspek	Void Method	Value-Returning Method
Penggunaan	Aksi/proses	Menghasilkan nilai
Bisa disimpan variabel	Tidak	Bisa

♦ 3. Contoh Perbandingan

```
// Void
void CetakJumlah(int a, int b)
{
    Console.WriteLine(a + b);
}
```

```
// Return
int HitungJumlah(int a, int b)
{
    return a + b;
}
```

Pemanggilan:

```
CetakJumlah(3, 4); // langsung tampil
```

```
int hasil = HitungJumlah(3, 4);
Console.WriteLine(hasil);
```

♦ 4. Kapan Menggunakan?

- Gunakan **void** → jika hanya ingin menjalankan aksi (misalnya print)
- Gunakan **return** → jika butuh hasil untuk diproses lagi

● 15.6 Method Overloading: Konsep dan Implementasi

Method Overloading adalah salah satu konsep inti dalam OOP yang memungkinkan sebuah kelas memiliki beberapa metode dengan **nama yang sama**, tetapi memiliki **parameter yang berbeda** (jumlah, tipe, atau urutan). Ini adalah bagian dari *Polymorphism* statis (compile-time), di mana compiler menentukan metode mana yang akan dijalankan berdasarkan "tanda tangan" (*signature*) metode yang dipanggil.

1. Aturan Main Method Overloading

Untuk melakukan overloading, metode harus berbeda dalam hal:

1. **Jumlah parameter** (misal: satu parameter vs dua parameter).
 2. **Tipe data parameter** (misal: int vs double).
 3. **Urutan parameter** (misal: (int, string) vs (string, int)).
- Mengubah **Tipe Return** (seperti mengubah void menjadi int) saja **tidak dianggap** sebagai overloading dan akan menyebabkan error jika parameter-parameternya tetap sama.

2. Analogi dalam Kehidupan Nyata

Bayangkan metode bernama Bayar().

- Jika Anda memberikan **Uang Tunai**, prosesnya adalah menerima uang dan memberi kembalian.
- Jika Anda memberikan **Kartu Kredit**, prosesnya adalah menggesek kartu dan meminta PIN.
- Jika Anda memberikan **QR Code**, prosesnya adalah memindai kode lewat aplikasi. Namanya tetap satu: Bayar, tapi caranya berbeda tergantung apa yang Anda berikan (parameternya).

3. Implementasi dalam C#

Berikut adalah contoh kelas Kalkulator yang menggunakan overloading untuk menjumlahkan berbagai jenis input:

```

1 using System;
2
3 namespace
4 {
5     // Metode 1: Menjumlahkan dua angka int
6     //reference
7     public int Tambah(int a, int b)
8     {
9         return a + b;
10    }
11
12    // Metode 2: Menjumlahkan tiga angka int (Beda jumlah parameter)
13    //reference
14    public int Tambah(int a, int b, int c)
15    {
16        return a + b + c;
17    }
18
19    // Metode 3: Menjumlahkan dua angka double (Beda tipe data)
20    //reference
21    public double Tambah(double a, double b)
22    {
23        return a + b;
24    }
25 }
26
27 namespace
28 {
29     //reference
30     public static void Main()
31     {
32         Kalkulator kal = new Kalkulator();
33
34         Console.WriteLine(kal.Tambah(5, 10)); // Memanggil Metode 1
35         Console.WriteLine(kal.Tambah(5, 10, 15)); // Memanggil Metode 2
36         Console.WriteLine(kal.Tambah(5.5, 10.5)); // Memanggil Metode 3
37     }
38 }

```

4. Manfaat Method Overloading

- **Keterbacaan Kode:** Anda tidak perlu menciptakan nama metode yang aneh-aneh seperti TambahInt(), TambahDouble(), atau TambahTigaAngka(). Cukup satu nama: Tambah().
- **Konsistensi:** Memudahkan programmer lain menggunakan kelas Anda karena mereka hanya perlu mengingat satu nama aksi dasar.
- **Fleksibilitas:** Memungkinkan objek untuk menangani berbagai jenis data input dengan cara yang paling sesuai.

5. Perbedaan dengan Method Overriding

Jangan sampai tertukar antara **Overloading** dan **Overriding**:

Fitur	Overloading	Overriding
Relasi	Terjadi dalam satu kelas yang sama.	Terjadi antara Parent dan Child class.
Nama Metode	Harus Sama.	Harus Sama.
Parameter	Harus Berbeda.	Harus Sama.

Fitur	Overloading	Overriding
Waktu Eksekusi	Compile-time.	Runtime.

- **15.7 Scope Variabel dalam Metode**

Scope Variabel (Cakupan Variabel) dalam metode merujuk pada wilayah di mana sebuah variabel "hidup" dan dapat diakses. Di C#, variabel tidak tersedia di mana-mana; mereka memiliki batas-batas tertentu yang ditentukan oleh blok kode (biasanya ditandai dengan kurung kurawal { }).

Memahami scope sangat penting untuk menghindari error seperti *"The name 'x' does not exist in the current context"* atau penggunaan variabel yang tidak sengaja tertukar.

1. Variabel Lokal (Local Variables)

Variabel yang dideklarasikan di dalam sebuah metode disebut **variabel lokal**.

Variabel ini hanya "dikenal" oleh kode yang berada di dalam metode tersebut. Begitu metode selesai dijalankan, variabel ini akan dihapus dari memori.

C#

```
public void Hitung()
{
    int angka = 10; // Ini variabel lokal
    Console.WriteLine(angka);
}

public void Cetak()
{
    // Console.WriteLine(angka); // ERROR: 'angka' tidak dikenal di sini
}
```

2. Block Scope (Cakupan Blok)

Variabel yang dideklarasikan di dalam blok kontrol seperti if, for, atau while memiliki cakupan yang lebih sempit lagi. Mereka hanya bisa diakses di dalam kurung kurawal tempat mereka dibuat.

C#

```
public void CekStatus(int nilai)
{
    if (nilai > 75)
    {
        string pesan = "Lulus"; // Hanya hidup di dalam blok 'if'
        Console.WriteLine(pesan);
    }

    // Console.WriteLine(pesan); // ERROR: pesan sudah "mati" di luar blok 'if'
}
```

3. Parameter Scope

Parameter yang Anda tulis di dalam tanda kurung metode (misal: void Simpan(string nama)) berperilaku seperti variabel lokal. Mereka bisa diakses di seluruh bagian dalam metode tersebut, tetapi tidak bisa diakses dari luar.

4. Konflik Nama: Field vs Local Variable

C# mengizinkan Anda memiliki variabel lokal dengan nama yang sama dengan **Field** (variabel di tingkat class). Namun, variabel lokal akan "menyembunyikan" field tersebut. Di sinilah kata kunci `this` menjadi pahlawan kita.

C#

```
public class Robot
{
    string nama = "Unit-A"; // Field (Class Scope)

    public void GantiNama(string nama) // Parameter 'nama'
    {
        // 'nama' merujuk ke parameter terdekat
        Console.WriteLine("Nama baru: " + nama);

        // 'this.nama' merujuk ke field milik class
        Console.WriteLine("Nama lama: " + this.nama);
    }
}
```

- **15.8 Recursive Methods: Konsep dan Contoh**

Recursive Method (Metode Rekursif) adalah sebuah teknik di dalam pemrograman di mana sebuah metode memanggil dirinya sendiri untuk menyelesaikan sebuah masalah.

Bayangkan Anda sedang berdiri di antara dua cermin yang berhadapan; Anda akan melihat bayangan di dalam bayangan secara terus-menerus. Dalam pemrograman, rekursi bekerja dengan cara memecah masalah besar menjadi masalah-masalah yang lebih kecil yang serupa, hingga mencapai titik paling sederhana.

1. Dua Syarat Wajib Rekursi

Agar sebuah metode rekursif tidak berjalan selamanya (yang akan menyebabkan *Stack Overflow Error*), ia harus memiliki dua komponen utama:

1. Base Case (Kondisi Berhenti): Titik di mana metode berhenti memanggil dirinya sendiri. Ini adalah solusi untuk bagian terkecil dari masalah.
 2. Recursive Step (Langkah Rekursif): Bagian di mana metode memanggil dirinya sendiri dengan argumen yang lebih kecil atau lebih sederhana, mendekati *Base Case*.
-

2. Contoh Klasik: Faktorial

Faktorial dari n (ditulis $n!$) adalah perkalian semua bilangan bulat positif dari n sampai 1.

Contoh: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

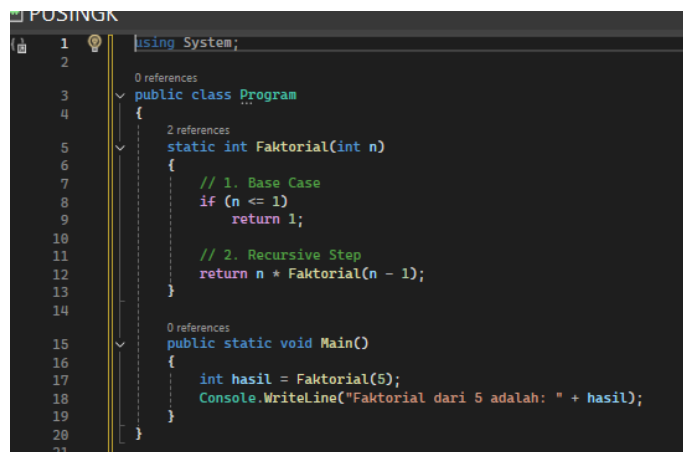
Logika Rekursifnya:

$5!$ adalah $5 \times 4!$

$4!$ adalah $4 \times 3!$

... dan seterusnya, sampai $1! = 1$ (Base Case).

Implementasi dalam C#:



```
1 using System;
2
3 public class Program
4 {
5     static int Faktorial(int n)
6     {
7         // 1. Base Case
8         if (n <= 1)
9             return 1;
10
11         // 2. Recursive Step
12         return n * Faktorial(n - 1);
13     }
14
15     public static void Main()
16     {
17         int hasil = Faktorial(5);
18         Console.WriteLine("Faktorial dari 5 adalah: " + hasil);
19     }
20 }
21
```

Bab 16: Exception Handling

• 16.1 Konsep Exception dan Error Handling

Dalam pengembangan aplikasi dengan C#, Exception Handling adalah mekanisme untuk menangani kesalahan yang muncul saat program berjalan (*runtime errors*). Tanpa penanganan ini, aplikasi Anda akan langsung *crash* dan berhenti berfungsi ketika menemui masalah (seperti pembagian nol atau file tidak ditemukan).

1. Struktur Dasar: Try-Catch-Finally

Di C#, penanganan eksepsi dilakukan menggunakan blok kode berikut:

- try: Blok kode yang berisi perintah yang "berisiko" menimbulkan error.
- catch: Blok yang akan dijalankan hanya jika terjadi error di dalam blok try.

- finally: Blok yang selalu dijalankan, baik terjadi error maupun tidak. Biasanya digunakan untuk membersihkan sumber daya (menutup database/file).

Contoh Sederhana:

```

1  using System;
2  try
3  {
4      int angka = 10;
5      int pembagi = 0;
6      int hasil = angka / pembagi; // Ini akan memicu DivideByZeroException
7  }
8  catch (DivideByZeroException ex)
9  {
10     Console.WriteLine("Kesalahan: Anda tidak bisa membagi angka dengan nol!");
11 }
12 finally
13 {
14     Console.WriteLine("Proses selesai.");
15 }

```

2. Hierarki Exception

Semua eksepsi di C# diturunkan dari kelas induk bernama System.Exception. Beberapa jenis eksepsi yang umum ditemui antara lain:

- NullReferenceException: Mencoba mengakses objek yang masih null.
- IndexOutOfRangeException: Mengakses indeks array yang tidak ada.
- FileNotFoundException: Mencoba membuka file yang tidak ditemukan di disk.
- FormatException: Gagal mengubah string menjadi tipe data lain (misal: "abc" diubah ke int).

3. Multiple Catch (Penanganan Spesifik)

Anda bisa memiliki lebih dari satu blok catch untuk menangani berbagai jenis error secara berbeda. C# akan menjalankan blok catch yang paling cocok dari atas ke bawah.

```

1  try
2  {
3      // Kode yang berisiko
4  }
5  catch (FileNotFoundException ex)
6  {
7      Console.WriteLine("File tidak ditemukan!");
8  }
9  catch (Exception ex) // Catch umum (sapu jagat)
10 {
11     Console.WriteLine("Terjadi error yang tidak terduga: " + ex.Message);
12 }

```

4. Menggunakan Kata Kunci throw

Terkadang, Anda ingin memicu eksepsi secara sengaja jika suatu kondisi bisnis tidak terpenuhi. Gunakan kata kunci throw.

C#

```
public void CekUmur(int umur)

{

    if (umur < 18)

    {

        throw new Exception("Maaf, Anda harus berusia di atas 18 tahun.");

    }

}
```

- **16.2 Blok try, catch, finally**

Dalam C#, blok try-catch-finally adalah fondasi dari manajemen kesalahan. Bayangkan blok ini sebagai "jaring pengaman" yang mencegah aplikasi Anda hancur berantakan saat terjadi sesuatu yang tidak terduga.

Berikut adalah penjelasan mendalam mengenai peran masing-masing blok:

1. Blok try: "Laboratorium Eksperimen"

Di dalam blok ini, Anda meletakkan kode yang berpotensi menyebabkan error. Program akan mencoba menjalankan baris demi baris. Jika terjadi error di salah satu baris, eksekusi di dalam blok try akan langsung dihentikan dan dilempar ke blok catch.

C#

```
try
{
    // Kode berisiko: Mengakses file yang mungkin tidak ada
    string konten = File.ReadAllText("data.txt");
    Console.WriteLine(konten);
}
```

2. Blok catch: "Unit Gawat Darurat"

Blok ini hanya akan berjalan jika terjadi kesalahan di dalam blok try. Di sini Anda menangkap objek *Exception* untuk mencari tahu apa yang salah dan memberikan solusi (seperti menampilkan pesan error yang ramah pengguna atau mencatatnya ke log).

C#

```
catch (FileNotFoundException ex)
{
    // Menangani error spesifik jika file tidak ditemukan
    Console.WriteLine("Maaf, file data.txt tidak ditemukan di folder ini.");
}
catch (Exception ex)
{
}
```

```
// Menangani error umum lainnya
Console.WriteLine("Terjadi kesalahan sistem: " + ex.Message);
}
```

3. Blok finally: "Petugas Kebersihan"

Ini adalah blok yang sangat istimewa. Kode di dalam finally pasti akan dijalankan, tidak peduli apakah terjadi error atau tidak. Gunanya adalah untuk "bersih-bersih", seperti menutup koneksi database atau menutup file agar memori tidak bocor.

4. Alur Kerja (Workflow)

Mari kita lihat skenario dunia nyata saat mencoba melakukan pembagian angka:

Skenario	Apa yang Terjadi?
Input Valid (10/2)	try jalan $\rightarrow$ catch dilewati $\rightarrow$ finally jalan.
Input Error (10/0)	try berhenti $\rightarrow$ catch menangkap error $\rightarrow$ finally jalan.

5. Contoh Implementasi Lengkap

C#

```
using System;
using System.IO;
```

```
class Program
```

```
{
    static void Main()
    {
        StreamReader reader = null;
        try
        {
            reader = new StreamReader("catatan.txt");
            Console.WriteLine(reader.ReadToEnd());
        }
        catch (IOException e)
        {
            Console.WriteLine("Gagal membaca file: " + e.Message);
        }
        finally
        {
            // Menutup file di sini sangat penting agar file tidak "terkunci"
            if (reader != null)
            {
                reader.Close();
                Console.WriteLine("File berhasil ditutup dengan aman.");
            }
        }
    }
}
```

```

    }
  }
}

```

- **16.3 Multiple Catch Blocks dan Exception Filtering**

Dalam situasi dunia nyata, satu blok kode try mungkin saja memicu lebih dari satu jenis kesalahan. Untuk menangani berbagai skenario ini secara spesifik, kita menggunakan **Multiple Catch Blocks**.

1. Konsep Multiple Catch Blocks

Konsep ini memungkinkan kita membuat beberapa "jaring" untuk menangani pengecualian yang berbeda dengan cara yang berbeda pula.

Aturan Penting: Urutan Catch

Saat menggunakan multiple catch, urutan itu sangat menentukan. Anda harus menangkap **Exception yang lebih spesifik (Subclass)** terlebih dahulu, baru kemudian **Exception yang lebih umum (Superclass)**.

Jika Anda menaruh catch (Exception e) (induk dari semua exception) di paling atas, maka blok catch di bawahnya tidak akan pernah dijalankan karena semua kesalahan sudah terserap oleh si "Induk".

Contoh Implementasi (Java):

```

Java
try {
    int[] angka = {1, 2, 3};
    int hasil = angka[1] / 0; // Berpotensi ArithmeticException
    System.out.println(angka[5]); // Berpotensi ArrayIndexOutOfBoundsException

} catch (ArithmeticException e) {
    System.out.println("Kesalahan Matematika: Tidak bisa membagi dengan nol.");

} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Kesalahan Array: Indeks tidak ditemukan.");

} catch (Exception e) {
    // Menangkap semua jenis kesalahan lainnya (General)
    System.out.println("Terjadi kesalahan sistem: " + e.getMessage());
}

```

2. Multi-Catch (Java 7+)

Jika Anda ingin menangani beberapa jenis exception dengan **logika yang sama**, Anda tidak perlu menulis blok catch terpisah. Anda bisa menggunakan operator pipa (|).

```

Java
try {
    // Kode berisiko
} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
    // Satu penanganan untuk dua jenis kesalahan
}

```

```
System.out.println("Terjadi kesalahan input atau perhitungan.");  
}
```

3. Exception Filtering

Exception Filtering adalah teknik untuk menyaring exception bukan hanya berdasarkan tipenya, tetapi juga berdasarkan kondisi tertentu (misalnya kode error atau pesan di dalamnya).

Di beberapa bahasa seperti C# atau VB.NET, terdapat kata kunci `when` untuk melakukan ini. Di Java, kita melakukannya secara manual menggunakan logika `if-else` di dalam blok `catch`.

Contoh Filtering di Java:

```
Java  
try {  
    // Operasi Database atau File  
} catch (IOException e) {  
    if (e.getMessage().contains("Access Denied")) {  
        System.out.println("Masalah Izin: Anda tidak punya akses.");  
    } else {  
        System.out.println("Masalah IO lainnya: " + e.getMessage());  
    }  
}
```

- **16.4 Throwing Exception dengan `throw`**

Dalam C#, kata kunci `throw` digunakan untuk memicu sebuah **Exception** secara sengaja. Jika blok `try-catch` digunakan untuk **menangkap** kesalahan yang jatuh, maka `throw` adalah tindakan **melempar** kesalahan tersebut karena ada aturan yang dilanggar.

Biasanya, kita menggunakan `throw` ketika logika bisnis tidak terpenuhi, meskipun secara teknis kode tersebut tidak menyebabkan aplikasi berhenti mendadak (*crash*).

1. Mengapa Kita Perlu Melempar Exception?

Tidak semua kesalahan berasal dari kegagalan sistem (seperti pembagian nol). Seringkali kesalahan berasal dari **aturan bisnis** yang tidak ditaati.

- **Contoh Kasus:** Seorang pengguna mencoba menarik uang Rp1.000.000, padahal saldonya hanya Rp50.000. Secara matematika, saldo bisa saja menjadi minus, tetapi secara aturan perbankan, ini adalah kesalahan yang harus "dilempar" untuk menghentikan transaksi.

2. Cara Menggunakan `throw`

Anda bisa melempar kelas `Exception` bawaan C# (seperti `ArgumentException`) atau pesan kustom Anda sendiri.

```
C#  
public void SetUmur(int umur)  
{  
    if (umur < 0)  
    {
```

```

        // Melempar exception karena umur tidak mungkin negatif
        throw new ArgumentOutOfRangeException("umur", "Umur tidak boleh kurang
dari nol.");
    }
    Console.WriteLine("Umur berhasil diatur.");
}

```

3. Re-throwing: Melempar Kembali Exception

Kadang-kadang, Anda menangkap exception di blok catch, melakukan sesuatu (seperti mencatat log kesalahan ke database), lalu ingin melemparnya kembali agar ditangani oleh bagian program yang lebih tinggi.

Ada dua cara untuk melakukan ini, dan perbedaannya sangat krusial:

- **throw; (Direkomendasikan):** Melempar kembali exception asli tanpa menghilangkan jejak asal-usul error (*Stack Trace*). Ini menjaga informasi baris mana yang pertama kali error.
- **throw ex; (Kurang Baik):** Melempar kembali tetapi menganggap error tersebut seolah-olah baru saja terjadi di baris throw tersebut, sehingga jejak error aslinya hilang.

```

C#
try
{
    // Kode berisiko
}
catch (Exception ex)
{
    Console.WriteLine("Mencatat log kesalahan...");
    throw; // Melempar ulang dengan aman
}

```

4. Perbedaan throw vs return

Fitur	return	throw
Tujuan	Mengirimkan hasil fungsi secara normal.	Menghentikan alur karena kondisi tidak normal.
Arah Alur	Kembali ke pemanggil metode tersebut.	Mencari blok catch terdekat ke atas.
Dampak	Metode selesai dengan sukses.	Metode dianggap gagal dan berhenti seketika.

5. Contoh Implementasi: Validasi Login

```

C#
public void ProsesLogin(string username, string password)

```

```

{
    if (string.IsNullOrEmpty(username))
    {
        throw new ArgumentException("Username tidak boleh kosong!");
    }

    if (password.Length < 8)
    {
        throw new Exception("Keamanan rendah: Password minimal 8 karakter.");
    }
}

```

- 16.5 Custom Exception Classes

Custom Exception Classes adalah kelas pengecualian yang Anda buat sendiri untuk menangani kondisi error yang spesifik bagi logika bisnis aplikasi Anda. Meskipun C# menyediakan banyak kelas exception bawaan, terkadang Anda membutuhkan nama error yang lebih deskriptif, seperti `SaldoTidakCukupException` atau `UserBelumVerifikasiException`. Membuat Custom Exception membuat kode Anda lebih profesional, lebih mudah dibaca, dan memudahkan proses *debugging*.

1. Aturan Dasar Membuat Custom Exception

Untuk membuat kelas exception sendiri, Anda harus mengikuti standar industri berikut:

1. **Inheritance:** Harus mewarisi dari kelas `System.Exception` (atau kelas turunannya).
 2. **Naming:** Nama kelas harus diakhiri dengan kata **"Exception"**.
 3. **Constructors:** Disarankan menyediakan minimal tiga konstruktor standar agar kompatibel dengan berbagai cara pelemparan error.
-

2. Implementasi Kode di C#

Berikut adalah cara membuat dan menggunakan Custom Exception untuk sistem perbankan:

C#

using System;

// 1. Mendefinisikan Custom Exception

```

public class SaldoTidakCukupException : Exception
{

```

```

    // Konstruktor kosong

```

```

    public SaldoTidakCukupException() { }

```

```

    // Konstruktor dengan pesan kustom

```

```

    public SaldoTidakCukupException(string message) : base(message) { }

```

```

    // Konstruktor dengan pesan kustom dan inner exception (error penyebab)

```

```

    public SaldoTidakCukupException(string message, Exception inner) :
    base(message, inner) { }

```

```

}

public class RekeningBank
{
    public decimal Saldo { get; private set; } = 50000;

    public void TarikTunai(decimal jumlah)
    {
        if (jumlah > Saldo)
        {
            // 2. Melempar Custom Exception
            throw new SaldoTidakCukupException($"Penarikan sebesar {jumlah} gagal. Saldo Anda hanya {Saldo}.");
        }
        Saldo -= jumlah;
    }
}

```

3. Cara Menangkap Custom Exception

Di sisi pengguna kode (Main program), Anda bisa menangkap error ini secara spesifik:

C#

```

public class Program
{
    public static void Main()
    {
        RekeningBank akun = new RekeningBank();

        try
        {
            akun.TarikTunai(100000);
        }
        catch (SaldoTidakCukupException ex)
        {
            // Menangani error spesifik bisnis kita
            Console.WriteLine("LOGIKA BANK: " + ex.Message);
        }
        catch (Exception ex)
        {
            // Menangani error umum lainnya
            Console.WriteLine("ERROR SISTEM: " + ex.Message);
        }
    }
}

```

4. Menambahkan Data Tambahan pada Exception

Kelebihan dari Custom Exception adalah Anda bisa menambahkan properti tambahan untuk memberikan informasi ekstra bagi tim pengembang atau sistem log.

```

C#
public class UserValidationException : Exception
{
    public string NamaUser { get; }
    public DateTime WaktuKejadian { get; }

    public UserValidationException(string message, string username) : base(message)
    {
        NamaUser = username;
        WaktuKejadian = DateTime.Now;
    }
}

```

- 16.6 Best Practices dalam Exception Handling

Menguasai *Exception Handling* bukan hanya tentang tahu cara menggunakan try-catch, tapi tentang menjaga aplikasi Anda tetap stabil, berperforma tinggi, dan mudah dilacak saat terjadi error.

Berikut adalah panduan **Best Practices** (praktik terbaik) dalam menangani eksepsi di C#:

1. Hanya Tangkap Eksepsi yang Bisa Anda Tangani

Jangan menangkap eksepsi jika Anda hanya akan "menelannya" atau mendiampkannya. Jika Anda tidak tahu cara memulihkan program dari error tersebut, biarkan eksepsi mengalir ke level yang lebih tinggi (atau biarkan aplikasi *crash* agar bug segera terdeteksi).

- **Buruk:** catch (Exception) { } (Menyembunyikan bug).
- **Baik:** Menangkap eksepsi, mencatat log, dan mungkin memberikan pesan yang masuk akal kepada pengguna.

2. Gunakan Eksepsi Spesifik, Bukan Umum

Selalu gunakan kelas eksepsi yang paling spesifik. Ini membantu Anda memahami penyebab error tanpa harus membaca detail pesan teksnya.

- **Hindari:** catch (Exception ex) di setiap blok.
- **Gunakan:** catch (FileNotFoundException), catch (UnauthorizedAccessException), dll.

3. Jangan Gunakan Eksepsi untuk Alur Program Biasa

Melempar (*throwing*) eksepsi sangat membebani CPU dan memori. Jangan gunakan eksepsi untuk logika yang sering terjadi atau bisa diprediksi.

- **Contoh Salah:** Menggunakan try-catch untuk mengecek apakah sebuah string adalah angka.
- **Contoh Benar:** Gunakan metode TryParse.

C#

// Lebih cepat dan bersih

```
if (int.TryParse(input, out int result)) { /* jalankan logika */ }
```

4. Gunakan throw; Bukan throw ex;

Saat Anda ingin melempar kembali eksepsi yang ditangkap di blok catch, gunakan kata kunci throw; saja. Ini akan menjaga **Stack Trace** (jejak baris kode yang error) tetap utuh.

```
C#
catch (Exception ex)
{
    Log(ex);
    throw; // Jejak asal error tetap aman
}
```

5. Prioritaskan Blok using atau finally

Pastikan sumber daya seperti koneksi database, *file stream*, atau *network socket* selalu ditutup, bahkan jika terjadi error. Blok using adalah cara paling elegan untuk melakukan ini secara otomatis.

```
C#
using (var stream = File.OpenRead("data.txt"))
{
    // Jika di sini error, file akan tetap ditutup otomatis
}
```

6. Berikan Pesan yang Bermakna (Tapi Aman)

- **Untuk Log:** Simpan detail teknis selengkap mungkin (*Stack Trace*, *Inner Exception*).
- **Untuk Pengguna:** Berikan pesan yang ramah dan tidak membingungkan. **Jangan** tampilkan detail teknis seperti nama server atau *query* database kepada pengguna karena berisiko dari sisi keamanan.

Ringkasan Tabel Strategi

Situasi	Tindakan Terbaik
Error yang bisa diprediksi	Gunakan if-else atau Try-Parse.
Pembersihan Sumber Daya	Gunakan blok using atau finally.
Logika Bisnis Dilanggar	Lempar Custom Exception.
Error Tidak Dikenal	Tangkap di level global (App Domain) dan catat log.



BAGIAN 4: PEMROGRAMAN BERORIENTASI OBJEK (OOP)

Bab 17: Pengenalan OOP dalam C#

- **17.1 Konsep Dasar OOP: Class, Object, Encapsulation, Inheritance, Polymorphism, Abstraction**

1. Class & Object (Cetak Biru & Hasil Produksi)

- **Class:** Sebuah *blueprint* atau cetak biru yang mendefinisikan atribut (data) dan perilaku (metode). Class tidak nyata, ia hanya rancangan.
- **Object:** Realisasi atau instansi nyata dari sebuah Class. Jika Class adalah "Denah Rumah", maka Object adalah "Rumah Fisik" yang Anda tempati.

2. Encapsulation (Pembungkusan)

Konsep untuk menyembunyikan data sensitif dan membatasi akses langsung dari luar. Kita menggunakan *access modifier* seperti *private* dan menyediakan akses melalui metode *getter* dan *setter*.

- **Analogi:** Seperti jam tangan. Anda bisa melihat waktu dan mengatur jarum jam (lewat tombol), tapi Anda tidak bisa (dan tidak perlu) menyentuh mesin di dalamnya secara langsung.

3. Inheritance (Pewarisan)

Memungkinkan sebuah kelas (Subclass) untuk mewarisi atribut dan metode dari kelas lain (Superclass). Ini sangat berguna untuk **reusability** (penggunaan kembali kode).

- **Contoh:** Class Kendaraan punya atribut kecepatan. Class Mobil dan Motor mewarisi Kendaraan, sehingga otomatis punya kecepatan tanpa perlu menulis ulang kodenya.

4. Polymorphism (Banyak Bentuk)

Kemampuan suatu objek untuk memiliki banyak bentuk. Biasanya terjadi melalui *Method Overriding* (mengubah isi metode warisan) atau *Method Overloading*.

- **Contoh:** Class Hewan punya metode bersuara(). Jika objeknya Kucing, suaranya "Meong". Jika Anjing, suaranya "Guk Guk". Metodenya sama (bersuara), tapi perilakunya berbeda.

5. Abstraction (Abstraksi)

Proses menyembunyikan detail implementasi yang rumit dan hanya menampilkan fungsi-fungsi penting kepada pengguna. Dalam kode, ini diimplementasikan menggunakan *abstract class* atau *interface*.

- **Analogi:** Saat Anda menyalakan TV, Anda hanya menekan tombol Power. Anda tidak perlu tahu bagaimana aliran listrik memicu panel LCD untuk menghasilkan gambar. Itu adalah detail teknis yang "diabstraksikan".

Ringkasan Perbedaan

Pilar	Fokus Utama
Encapsulation	Keamanan dan pengorganisasian data (Data Hiding).
Inheritance	Penghematan kode dan struktur hierarki (Reusability).
Polymorphism	Fleksibilitas aksi yang berbeda dengan nama yang sama.
Abstraction	Penyederhanaan kompleksitas (fokus pada <i>apa</i> bukan <i>bagaimana</i>).

• 17.2 Perbandingan Prosedural vs OOP

Memilih antara pemrograman **Prosedural** dan **OOP (Object-Oriented Programming)** seperti memilih antara menulis instruksi resep masakan selangkah demi selangkah (Prosedural) atau membangun sebuah dapur otomatis di mana setiap alat tahu tugasnya masing-masing (OOP).

Berikut adalah perbandingan mendalam untuk membantu Anda memahami kapan harus menggunakan salah satunya:

1. Perbedaan Konsep Dasar

Fitur	Pemrograman Prosedural	Pemrograman Berorientasi Objek (OOP)
Fokus Utama	Tugas/Fungsi. Menekankan pada urutan langkah atau algoritma.	Data/Objek. Menekankan pada benda yang memiliki data dan perilaku.
Pendekatan	<i>Top-Down approach</i> (dari atas ke bawah).	<i>Bottom-Up approach</i> (dari bawah ke atas).
Unit Dasar	Fungsi atau Prosedur.	Objek dan Kelas.
Keamanan Data	Kurang aman; data biasanya bersifat global dan bisa diakses siapa saja.	Sangat aman; data bisa disembunyikan menggunakan <i>Encapsulation</i> .

2. Analogi Kasus: Sistem Penggajian

Pendekatan Prosedural

Program akan berjalan secara linier:

1. Ambil data karyawan dari database.
2. Hitung gaji pokok berdasarkan jam kerja.
3. Hitung potongan pajak.
4. Cetak slip gaji.

Masalah muncul jika struktur data karyawan berubah; Anda harus mengubah semua fungsi yang berhubungan dengan data tersebut.

Pendekatan OOP

Anda membuat objek Karyawan:

1. Objek Karyawan memiliki data sendiri (nama, tarif jam).
2. Objek Karyawan memiliki metode sendiri (hitungGaji()).
3. Anda cukup memerintahkan: karyawan.hitungGaji().

Jika aturan gaji berubah, Anda cukup memperbaiki kode di dalam kelas Karyawan tanpa mengganggu bagian program lainnya.

3. Kelebihan dan Kekurangan

Prosedural

- **Kelebihan:** Sederhana untuk dipelajari, menggunakan lebih sedikit memori, dan sangat efisien untuk tugas-tugas kecil atau perhitungan matematis murni.
- **Kekurangan:** Sulit dikelola untuk proyek besar. Jika satu bagian berubah, sering kali terjadi efek domino yang merusak bagian lain (*spaghetti code*).

OOP

- **Kelebihan:** Sangat mudah dikelola untuk proyek skala besar, kode dapat digunakan kembali (*reusability*), dan memudahkan kolaborasi tim karena setiap orang bisa mengerjakan objek yang berbeda.
 - **Kekurangan:** Memerlukan pemikiran desain yang lebih matang di awal, cenderung lebih lambat (sedikit saja) karena overhead memori, dan kurva belajarnya lebih tajam.
-

4. Mana yang Harus Digunakan?

- **Gunakan Prosedural jika:** Anda sedang membuat skrip otomatisasi sederhana, algoritma matematika, atau program yang sangat kecil dan hanya sekali pakai. Bahasa populer: **C, Pascal, BASIC**.
 - **Gunakan OOP jika:** Anda membangun aplikasi yang kompleks, berorientasi pengguna (UI), atau sistem yang akan terus dikembangkan dalam waktu lama (seperti web, aplikasi mobile, atau game). Bahasa populer: **Java, C++, Python, C#**.
-

Bab 18: Class dan Object

- **18.1 Mendefinisikan Class**

1. Mendefinisikan Class di C#

Berbeda dengan Java, C# menggunakan konvensi **PascalCase** untuk nama metode dan memiliki fitur unik bernama **Properties**.

```
1 using System;
2
3 namespace AplikasiKeren
4 {
5     // references
6     public class Mobil
7     {
8         // Field (Data mentah)
9         private string _merek;
10
11         // Property (Encapsulation yang lebih elegan dibanding Getter/Setter manual)
12         // 1 reference
13         public string Merek
14         {
15             get { return _merek; }
16             set { _merek = value; }
17         }
18
19         // references
20         public int Kecepatan { get; set; } // Auto-implemented property
21
22         // Method
23         // 0 references
24         public void Klakson()
25         {
26             Console.WriteLine($"({Merek}) berbunyi: Beep Beep!");
27         }
28     }
29 }
```

2. Exception Handling (Try-Catch-When)

C# memiliki fitur **Exception Filter** menggunakan kata kunci **when**, yang memungkinkan Anda menyaring error lebih detail tanpa masuk ke dalam blok **if** di dalam **catch**.

```
10 }
11 {
12     Console.WriteLine("Koneksi gagal!");
13 }
14 {
15     Console.WriteLine($"[Error] Pesan: {e.Message}");
16 }
17 {
18     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
19 }
20 {
21     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
22 }
23 {
24     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
25 }
26 {
27     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
28 }
29 {
30     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
31 }
32 {
33     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
34 }
35 {
36     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
37 }
38 {
39     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
40 }
41 {
42     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
43 }
44 {
45     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
46 }
47 {
48     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
49 }
50 {
51     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
52 }
53 {
54     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
55 }
56 {
57     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
58 }
59 {
60     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
61 }
62 {
63     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
64 }
65 {
66     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
67 }
68 {
69     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
70 }
71 {
72     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
73 }
74 {
75     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
76 }
77 {
78     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
79 }
80 {
81     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
82 }
83 {
84     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
85 }
86 {
87     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
88 }
89 {
90     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
91 }
92 {
93     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
94 }
95 {
96     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
97 }
98 {
99     Console.WriteLine("Error: Berapa kali lagi akan mencoba?");
100 }
```

3. Fitur Unggulan C# dibanding Java

Jika Anda beralih dari Java ke C#, ada beberapa fitur "quality of life" yang akan Anda sukai:

- **LINQ (Language Integrated Query):** Cara super ringkas untuk memproses data/list seperti SQL.
 - *Contoh:* `var lulus = listSiswa.Where(s => s.Nilai > 75);`
- **Struct vs Class:** C# punya struct untuk objek ringan yang disimpan di *stack* (bukan *heap*), lebih efisien untuk koordinat (X, Y) atau warna.
- **Namespace:** Mirip dengan package di Java, tapi lebih fleksibel dalam struktur folder.
- **Properties:** Menghapus kebutuhan untuk menulis ribuan baris `getNama()` dan `setNama()`.

4. Perbandingan Cepat

Fitur	Java	C#
Metode Utama	public static void main(String[] args)	static void Main(string[] args)
Pewarisan	extends	:
Interface	implements	:
Cetak ke Konsol	System.out.println()	Console.WriteLine()

- **18.2 Membuat Instance Object**

1. Proses Instansiasi

Instansiasi adalah proses mengalokasikan memori untuk objek baru dan memanggil **Constructor** untuk menginisialisasi objek tersebut.

Sintaks Dasar:

C#

```
NamaClass namaObjek = new NamaClass();
```

Contoh Nyata:

Jika kita punya class Karyawan, cara membuat objeknya adalah:

C#

```
Karyawan karyawan1 = new Karyawan(); // Membuat instance baru
```

```
karyawan1.Nama = "Budi"; // Mengisi data ke objek
```

```
karyawan1.Bekerja(); // Memanggil aksi objek
```

2. Apa yang Terjadi di Balik Layar?

Saat baris `new NamaClass()` dieksekusi, ada dua hal utama yang terjadi di memori komputer:

1. **Alokasi Memori:** Sistem menyediakan ruang di **Heap Memory** untuk menyimpan data objek tersebut.
2. **Inisialisasi: Constructor** dijalankan untuk memberikan nilai awal pada atribut/field objek.

3. Menggunakan Constructor dengan Parameter

Seringkali kita ingin langsung memberikan data saat objek dibuat. Ini dilakukan dengan mendefinisikan constructor di dalam class.

```

C#
public class Robot
{
    public string Nama;

    // Constructor
    public Robot(string namaAwal)
    {
        Nama = namaAwal;
    }
}

// Membuat instance dengan data awal
Robot bot1 = new Robot("Optimus");
Console.WriteLine(bot1.Nama); // Output: Optimus

```

4. Perbedaan Instance vs Reference

Penting untuk dipahami bahwa variabel seperti bot1 sebenarnya adalah sebuah **Reference** (penunjuk) ke lokasi objek di memori, bukan objek itu sendiri.

```

C#
Robot bot2 = bot1; // bot2 sekarang menunjuk ke objek yang sama dengan bot1
bot2.Nama = "Megatron";

```

```

Console.WriteLine(bot1.Nama); // Output: Megatron (karena mereka menunjuk objek yang sama)

```

● 18.3 Multiple Objects dari Satu Class

Salah satu kekuatan utama OOP adalah **reusability**. Setelah Anda mendefinisikan satu class, Anda bisa membuat ribuan objek (**Multiple Objects**) yang berbeda-beda dari cetak biru tersebut. Setiap objek akan memiliki struktur yang sama, tetapi menyimpan data (state) yang unik bagi dirinya sendiri.

1. Konsep Objek Mandiri

Bayangkan class sebagai formulir kosong. Ketika Anda mengisi formulir tersebut, Anda membuat satu objek. Jika ada 10 orang mengisi formulir yang sama, akan ada 10 objek berbeda. Perubahan data pada objek A **tidak akan memengaruhi** data pada objek B.

2. Implementasi dalam C#

Berikut adalah contoh bagaimana kita membuat tiga objek berbeda dari satu class Hero dalam game:

```

1 using System;
2
3 // reference
4 public class Hero
5 {
6     public string Nama;
7     public int HP;
8
9     // reference
10    public void Serang()
11    {
12        Console.WriteLine($"{Nama} menyerang dengan sisa HP: {HP}>");
13    }
14
15 // reference
16 public class Program
17 {
18     // reference
19     public static void Main()
20     {
21         // Membuat Objek Pertama
22         Hero hero1 = new Hero();
23         hero1.Nama = "Arang";
24         hero1.HP = 100;
25
26         // Membuat Objek Kedua
27         Hero hero2 = new Hero();
28         hero2.Nama = "Legolas";
29         hero2.HP = 80;
30
31         // Membuat Objek Ketiga
32         Hero hero3 = new Hero();
33         hero3.Nama = "Gimli";
34         hero3.HP = 120;
35
36         // Memanggil metode masing-masing
37         hero1.Serang(); // Output: Arang menyerang dengan sisa HP: 100
38         hero2.Serang(); // Output: Legolas menyerang dengan sisa HP: 80
39         hero3.Serang(); // Output: Gimli menyerang dengan sisa HP: 120
40     }
41 }

```

3. Mengelola Banyak Objek dengan Array atau List

Jika Anda memiliki sangat banyak objek (misalnya 100 musuh dalam game), tidak efisien jika harus memberi nama variabel satu per satu (musuh1, musuh2, dst). Kita bisa menggunakan **List** untuk menampungnya.

```

1 using System.Collections.Generic;
2
3 // Membuat daftar untuk menampung banyak objek Hero
4 List<Hero> pasukan = new List<Hero>();
5
6 // Menambahkan objek ke dalam list
7 pasukan.Add(new Hero { Nama = "Prajurit A", HP = 50 });
8 pasukan.Add(new Hero { Nama = "Prajurit B", HP = 50 });
9
10 // Mengakses semua objek sekaligus menggunakan loop
11 foreach (var h in pasukan)
12 {
13     h.Serang();
14 }

```

● 18.4 Accessing Members: Fields dan Method

1. Mengakses Fields (Variabel)

Fields menyimpan status atau data dari sebuah objek. Anda bisa **membaca** nilainya atau **mengubah** nilainya selama *access modifier*-nya mengizinkan (biasanya public).

```

1 // Membuat objek
2 Mobil mobilSaya = new Mobil();
3
4 // MENGUBAH nilai Field (Write)
5 mobilSaya.warna = "Merah";
6 mobilSaya.kecepatan = 100;
7
8 // MEMBACA nilai Field (Read)
9 Console.WriteLine("Warna mobil: " + mobilSaya.warna);
10

```

2. Mengakses Methods (Fungsi)

Methods adalah aksi yang bisa dilakukan oleh objek. Saat memanggil metode, Anda harus menyertakan tanda kurung () meskipun tidak ada parameter yang dikirimkan.

C#

// Memanggil metode tanpa parameter

mobilSaya.StarterMesin();

// Memanggil metode dengan parameter

mobilSaya.TambahKecepatan(50);

3. Masalah Keamanan: Kenapa tidak semua Field public?

Dalam praktik profesional, mengakses field secara langsung (objek.field = nilai) dianggap kurang baik karena tidak ada filter. Itulah sebabnya kita menggunakan **Encapsulation**.

- **Field:** Biasanya dibuat private agar tidak bisa diubah sembarangan dari luar.
- **Property:** Digunakan sebagai gerbang untuk mengakses field tersebut dengan aman.

Contoh Perbandingan:

Cara Akses	Kode	Penjelasan
Langsung (Beresiko)	mobil.bensin = -500;	Data rusak karena bensin tidak mungkin negatif.
Lewat Property	mobil.Bensin = -500;	Di dalam properti, kita bisa beri logika: "Jika nilai < 0, set ke 0".

4. Contoh Lengkap dalam C#

Berikut adalah gambaran bagaimana sebuah objek diakses member-membernya:

```

1 using System;
2
3 2 references
4 public class KarakterGame
5 {
6     public string nama; // Field
7     private int _darah = 100; // Private Field
8
9     // Properti untuk mengakses _darah secara aman
10    2 references
11    public int Darah
12    {
13        get { return _darah; }
14        set { if (value >= 0) _darah = value; }
15    }
16
17    1 reference
18    public void Sapa() // Method
19    {
20        Console.WriteLine($"Halo, saya {nama}!");
21    }
22
23    0 references
24    public class Program
25    {
26        0 references
27        public static void Main()
28        {
29            KarakterGame player = new KarakterGame();
30
31            // 1. Akses Field
32            player.nama = "Joko";
33
34            // 2. Akses Property
35            player.Darah = 80;
36
37            // 3. Akses Method
38            player.Sapa();
39
40            Console.WriteLine($"Tipe darah {player.nama}: {player.Darah}");
41        }
42    }
43 }

```

Bab 19: Constructor dan Destructor

● 19.1 Default Constructor

1. Apa itu Default Constructor?

Secara teknis, Default Constructor adalah metode yang:

- Memiliki nama yang **sama persis** dengan nama class.
- **Tidak memiliki parameter** di dalam tanda kurungnya ().
- Tidak memiliki tipe pengembalian (*return type*), bahkan bukan void.

Fungsi utamanya: Memberikan nilai awal (default) pada field/properti objek saat kata kunci new dipanggil.

2. Constructor Otomatis (Implicit)

Salah satu keunikan C# adalah: jika Anda **tidak menulis constructor sama sekali**, compiler akan secara otomatis membuat satu default constructor kosong untuk Anda di belakang layar.

C#

```
public class Siswa
```

```
{
```

```
    public string Nama;
```

```
// Tidak ada constructor tertulis di sini

}
```

// Di Main program:

Siswa s = new Siswa(); // Ini tetap bisa jalan karena ada Default Constructor otomatis.

Pada tahap ini, semua field akan diisi dengan nilai default bahasa C#:

-numeric (int, double, dll) menjadi **0**.

-bool menjadi **false**.

-reference types (string, object) menjadi **null**.

3. Menulis Default Constructor Secara Eksplisit

Terkadang, Anda ingin objek memiliki nilai awal yang "lebih bermakna" daripada sekadar nol atau null. Di sinilah Anda mendefinisikannya sendiri.

```
1 public class AkunBank
2 {
3     public double Saldo;
4     public string Status;
5
6     // Menulis Default Constructor sendiri
7     public AkunBank()
8     {
9         Saldo = 100000; // Set saldo awal otomatis 100rb
10        Status = "Aktif";
11        Console.WriteLine("Objek AkunBank baru saja dibuat.");
12    }
13 }
```

4. Hati-Hati: "Aturan Hilangnya" Default Constructor

Ini adalah bagian yang sering menjebak pemula: **Jika Anda membuat constructor dengan parameter, maka compiler tidak akan lagi membuat Default Constructor otomatis.**

```
1 public class Laptop
2 {
3     public string Merk;
4
5     // Anda membuat constructor dengan parameter
6     public Laptop(string merk)
7     {
8         Merk = merk;
9     }
10 }
11
12 // DI SINI AKAN ERROR:
13 // Laptop l = new Laptop();
14 // Kenapa? Karena Default Constructor otomatis sudah "hilang" diganti oleh yang berparameter.
```

- **19.2 Parameterized Constructor**

Parameterized Constructor adalah constructor yang memiliki satu atau lebih parameter di dalam tanda kurungnya. Jika *Default Constructor* memberikan nilai standar, maka *Parameterized Constructor* memungkinkan Anda untuk menyuntikkan data spesifik langsung pada saat objek diciptakan.

Ini adalah cara yang paling efisien untuk memastikan sebuah objek memiliki data yang valid sejak detik pertama ia hidup di memori.

1. Sintaks dan Cara Kerja

Di C#, Anda mendefinisikan parameterized constructor dengan mencantumkan tipe data dan nama variabel di dalam kurung setelah nama class.

```
C#
public class Hero
{
    public string Nama;
    public string Role;

    // Parameterized Constructor
    public Hero(string namaAwal, string roleAwal)
    {
        Nama = namaAwal;
        Role = roleAwal;
    }
}
```

2. Mengapa Menggunakan Parameterized Constructor?

1. **Efisiensi:** Anda tidak perlu menulis banyak baris kode setelah new untuk mengisi field satu per satu.
2. **Keharusan (Mandatory Data):** Anda bisa memaksa agar objek tidak bisa dibuat tanpa data tertentu. Misalnya, tidak boleh ada objek User tanpa Username.
3. **Keterbacaan:** Kode menjadi lebih ringkas dan niat pembuatannya jelas.

3. Contoh Implementasi di C#

Mari bandingkan cara membuat objek tanpa dan dengan parameterized constructor:

Tanpa (Menggunakan Default):

```
C#
Mobil m = new Mobil();
m.Merk = "Toyota";
m.Tahun = 2022;
```

Dengan Parameterized Constructor:

```
C#
// Jauh lebih singkat dan padat
Mobil m = new Mobil("Toyota", 2022);
```

4. Penggunaan Kata Kunci this

Seringkali, programmer menggunakan nama parameter yang **sama persis** dengan nama field agar kode mudah dipahami. Dalam kondisi ini, kita menggunakan kata kunci **this** untuk membedakan mana yang merupakan field milik class dan mana yang merupakan parameter.

```

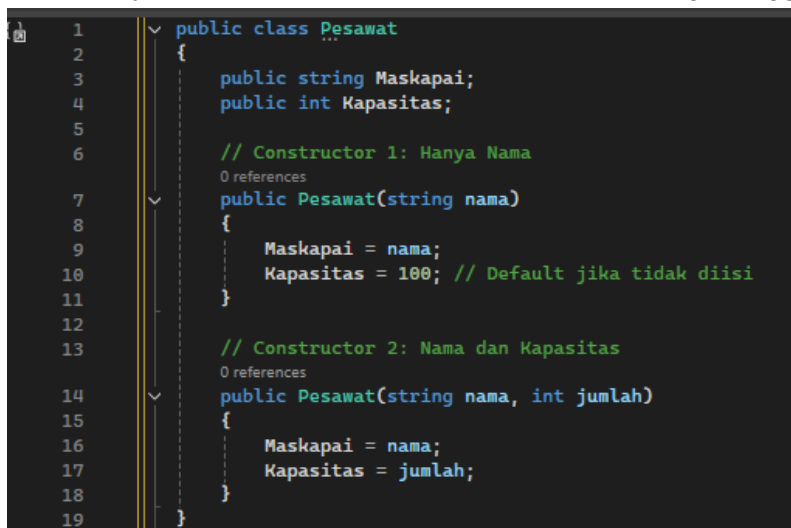
C#
public class Produk
{
    public string Nama;

    public Produk(string Nama)
    {
        // this.Nama merujuk ke Field di atas
        // Nama merujuk ke Parameter di dalam kurung
        this.Nama = Nama;
    }
}

```

5. Constructor Overloading

Anda bisa memiliki **lebih dari satu** constructor dalam satu class selama jumlah atau tipe datanya berbeda. Ini memberikan fleksibilitas bagi pengguna class Anda.



```

1  public class Pesawat
2  {
3      public string Maskapai;
4      public int Kapasitas;
5
6      // Constructor 1: Hanya Nama
7      public Pesawat(string nama)
8      {
9          Maskapai = nama;
10         Kapasitas = 100; // Default jika tidak diisi
11     }
12
13     // Constructor 2: Nama dan Kapasitas
14     public Pesawat(string nama, int jumlah)
15     {
16         Maskapai = nama;
17         Kapasitas = jumlah;
18     }
19 }

```

19.3 Constructor Overloading

Constructor Overloading adalah teknik di mana Anda mendefinisikan **lebih dari satu constructor** di dalam satu kelas yang sama. Tujuannya adalah untuk memberikan berbagai cara bagi pengguna kelas tersebut untuk menginisialisasi sebuah objek, tergantung pada data apa yang mereka miliki saat itu.

Konsep ini mengikuti aturan **Polymorphism** (polimorfisme statis), di mana compiler akan menentukan constructor mana yang akan dipanggil berdasarkan jumlah, tipe, atau urutan parameter yang dikirimkan.

1. Mengapa Menggunakan Constructor Overloading?

- **Fleksibilitas:** Memberikan opsi kepada pengguna kelas untuk membuat objek dengan detail lengkap atau hanya dengan data minimal.
- **Nilai Default:** Anda bisa mengatur nilai default secara otomatis jika parameter tertentu tidak diberikan.
- **Keterbacaan:** Membuat kode lebih bersih karena Anda tidak perlu memanggil banyak metode *setter* setelah membuat objek.

2. Implementasi dalam C#

Mari kita ambil contoh kelas Karyawan. Seorang karyawan bisa saja baru bergabung (hanya punya nama) atau sudah memiliki ID dan posisi tertentu.

```
1 using System;
2
3 public class Karyawan
4 {
5     public string Nama;
6     public string Posisi;
7     public int Gaji;
8
9     // Constructor 1: Tanpa parameter (Default)
10    // reference
11    public Karyawan()
12    {
13        Nama = "Tanpa Nama";
14        Posisi = "Intern";
15        Gaji = 0;
16    }
17
18    // Constructor 2: Hanya Nama (Overload 1)
19    // reference
20    public Karyawan(string nama)
21    {
22        Nama = nama;
23        Posisi = "Staff"; // Default posisi jika tidak diisi
24        Gaji = 5000000;
25    }
26
27    // Constructor 3: Nama, Posisi, dan Gaji (Overload 2)
28    // reference
29    public Karyawan(string nama, string posisi, int gaji)
30    {
31        Nama = nama;
32        Posisi = posisi;
33        Gaji = gaji;
34    }
35 }
```

3. Cara Penggunaan di Main Program

C#

```
public class Program
```

```
{
    public static void Main()
    {
        // Menggunakan Constructor 1
        Karyawan k1 = new Karyawan();

        // Menggunakan Constructor 2
        Karyawan k2 = new Karyawan("Budi");

        // Menggunakan Constructor 3
        Karyawan k3 = new Karyawan("Siska", "Manager", 15000000);
    }
}
```

4. Teknik "Constructor Chaining" dengan Kata Kunci this

Dalam praktik terbaik di C#, daripada mengulang-ulang penulisan kode inisialisasi di setiap constructor, kita bisa menggunakan **Constructor Chaining**. Ini memungkinkan satu constructor memanggil constructor lainnya di kelas yang sama.

```

1  public class Karyawan
2  {
3      public string Nama;
4      public string Posisi;
5
6      // Constructor lengkap
7      public Karyawan(string nama, string posisi)
8      {
9          this.Nama = nama;
10         this.Posisi = posisi;
11     }
12
13     // Constructor ini memanggil constructor di atas (Chaining)
14     public Karyawan(string nama) : this(nama, "Staff")
15     {
16         // Kode tambahan jika perlu
17     }
18 }

```

Constructor Chaining sangat berguna untuk menjaga prinsip **DRY (Don't Repeat Yourself)**. Jika ada logika inisialisasi yang berubah, Anda hanya perlu mengubahnya di satu tempat (constructor utama).

● 19.4 **this** Keyword

1. Membedakan Field dengan Parameter (Disambiguasi)

Ini adalah penggunaan yang paling umum. Ketika nama parameter di dalam *constructor* atau *method* sama persis dengan nama *field* di dalam kelas, compiler akan bingung. *this* digunakan untuk memperjelas bahwa yang dimaksud adalah variabel milik kelas.

C#

```

public class Siswa {
    private string nama;

    public void SetNama(string nama) {
        // nama (tanpa this) merujuk ke parameter
        // this.nama merujuk ke field milik class
        this.nama = nama;
    }
}

```

2. Constructor Chaining (Memanggil Constructor Lain)

Seperti yang sempat kita singgung pada pembahasan *overloading*, *this* digunakan untuk memanggil *constructor* lain dalam kelas yang sama. Ini membantu menghindari pengulangan kode (*redundancy*).

C#

```

public class Player {
    public string Nama;
    public int Level;

    // Constructor 1
    public Player(string nama, int level) {
        this.Nama = nama;
        this.Level = level;
    }
}

```

```
// Constructor 2 memanggil Constructor 1 menggunakan : this()
public Player(string nama) : this(nama, 1) {
    // Logika tambahan jika ada
}
}
```

3. Mengirim Objek Saat Ini sebagai Argumen

Terkadang, sebuah objek perlu mengirimkan dirinya sendiri ke metode di kelas lain agar kelas tersebut bisa memproses datanya.

C#

```
public class Logger {
    public void CetakInfo(Siswa s) {
        Console.WriteLine("Data Siswa: " + s.GetNama());
    }
}

public class Siswa {
    public void LaporkKeLogger() {
        Logger log = new Logger();
        log.CetakInfo(this); // Mengirimkan "saya sendiri" ke logger
    }
}
```

4. Membuat Extension Methods

Dalam C# tingkat lanjut, *this* digunakan pada parameter pertama sebuah metode statis untuk menjadikannya sebagai *Extension Method* (metode tambahan untuk tipe data tertentu).

C#

```
public static class StringExtensions {
    // Kata kunci 'this' di sini membuat semua string punya metode 'Sapa'
    public static void Sapa(this string str) {
        Console.WriteLine("Halo, " + str);
    }
}
```

// Penggunaan:
"Budi".Sapa();

● 19.5 Destructor dan Finalizer (Konsep Lanjutan)

1. Konsep Dasar Destructor/Finalizer

C# adalah bahasa yang memiliki **Garbage Collector (GC)**. Artinya, Anda tidak perlu menghapus objek secara manual (berbeda dengan bahasa C++). GC akan otomatis mendeteksi objek yang sudah tidak terikat pada variabel mana pun dan menghapusnya untuk membebaskan ruang memori.

Finalizer dipanggil secara otomatis oleh GC sesaat sebelum objek benar-benar dihancurkan.

2. Sintaks Penulisan

Sintaks destructor di C# mirip dengan C++, yaitu menggunakan tanda tilde (~) diikuti dengan nama class.

- Tidak memiliki *access modifier* (seperti public atau private).
- Tidak memiliki parameter.
- Tidak bisa dipanggil secara manual oleh kode Anda.

```
1  class KoneksiDatabase
2  {
3      // Constructor
4      0 references
5      public KoneksiDatabase()
6      {
7          Console.WriteLine("Koneksi dibuka.");
8      }
9      // Destructor (Finalizer)
10     0 references
11     ~KoneksiDatabase()
12     {
13         // Kode pembersihan di sini
14         Console.WriteLine("Koneksi ditutup otomatis oleh sistem.");
15     }
16 }
```

3. Kenapa Jarang Digunakan? (Penting!)

Di dunia .NET modern, penggunaan destructor/finalizer sebenarnya **sangat jarang disarankan** karena beberapa alasan:

1. **Ketidakpastian Waktu:** Anda tidak tahu *kapan* GC akan berjalan. Objek mungkin sudah tidak dipakai, tapi destructor baru jalan 10 menit kemudian. Ini berbahaya jika Anda ingin menutup file atau koneksi database secepat mungkin.
2. **Beban Performa:** Objek yang memiliki finalizer membutuhkan dua siklus Garbage Collection untuk benar-benar terhapus, sehingga membebani memori lebih lama.

4. Solusi Lebih Baik: Interface IDisposable

Alih-alih mengandalkan destructor yang waktunya tidak pasti, programmer C# menggunakan pola **IDisposable** dan blok **using**. Ini memungkinkan kita menghapus sumber daya (seperti file atau koneksi) secara instan dan sengaja.

```

1 public class AlatTulis : IDisposable
2 {
3     1 reference
4     public void Dispose()
5     {
6         // Tutup file atau hapus data sensitif SEKARANG
7         Console.WriteLine("Sumber daya dibersihkan secara manual.");
8     }
9
10    // Penggunaan yang direkomendasikan:
11    using (AlatTulis pen = new AlatTulis())
12    {
13        // Pakai objeknya
14        } // Begitu sampai sini, Dispose() otomatis dipanggil.
15

```

Perbandingan Ringkas

Fitur	Destructor / Finalizer	IDisposable (Dispose)
Pemicu	Otomatis oleh Garbage Collector.	Manual oleh programmer (atau using).
Waktu	Tidak terprediksi (Non-deterministic).	Detik itu juga (Deterministic).
Tujuan	"Jaring pengaman" terakhir.	Pembersihan sumber daya yang cepat.