

Project Otak : Seri Buku Komunitas

Knowledge
is **Free**



Indonesia.net
Developer Community

Pengenalan Bahasa C#

- Membahas sintak-sintak bahasa C#
- Object Oriented Programming pada C#
- Studi kasus OOP dalam C#



Disertai Code

Agus Kurniawan • Risman Adnan • Panji Aryaputra
Norman Sasono • Ali Ahmad Heryana • M Fathur Rahman
I Wayan Saryada • Adi Wirasta

Pengenalan
BAHASA C#
(CSH101)

**Agus Kurniawan
Risman Adnan
Panji Aryaputra
Norman Sasono
Ali Ahmad Heryana
M. Fathur Rahman
I Wayan Saryada
Adi Wirasta**

Project Otak
2004

Project Otak

Project otak adalah project open source yang bertujuan untuk menyediakan resources tentang informasi teknologi .NET bagi orang-orang yang ingin belajar teknologi .NET.

Trademark Acknowledgements

Team project otak akan berusaha menyediakan informasi trademark termasuk semua produk yang telah disebut didalam buku ini.

Windows, Framework .NET, Visual C#, dan Visual Studio.NET adalah trademark dari Microsoft

Credits

Project Manager

Agus Kurniawan

Technical Writer

Agus Kurniawan

Risman Adnan

Panji Aryaputra

Norman Sasono

Ali Ahmad Heryana

M. Fathur Rahman

I Wayan Saryada

Adi Wirasta

Editor

Agus Kurniawan

Risman Adnan

Cover Designer

Qmunk

Version 1.0

Printed: 29 October 2004

Book Code: CSH101

Update E-Book : <http://otak.csharpindonesia.net>

Semua materi yang ada didalam buku ini adalah satu kesatuan. Tidak boleh sebagian atau seluruh materi didalam buku ini diubah tanpa seijin team project otak.

Kata Pengantar

Atas rahmat dan hidayah dari Tuhan Yang Maha Esa, Project Otak yang berhasil menyelesaikan project pertamanya yaitu e-book dengan judul Pengenalan Bahasa C#. Semoga buku ini dapat bermanfaat bagi masyarakat Indonesia yang ingin belajar .NET dengan bahasa C#.

Jika ada kesalahan atau saran demi perbaikan kualitas buku ini maka bisa langsung menghubungi kami.

Selamat membaca buku C# ini.....dan tetap pada motto KNOWLEDGE is FREE.

Salam,
Jakarta, 29 Oktober 2004

Team Project Otak

Tentang Penulis



Agus Kurniawan

Agus Kurniawan adalah *HardCore Developer*, *Software Architect*, *Trainer*, *Writer* dan *Lecturer*. Lulusan dari Teknik Elektro ITS. Agus memulai coding dengan bahasa C/C++ pada mesin processor Intel 386. Tertarik pada bidang software architecture, networking dan distributed system, artificial intelligent. Sehari-harinya Agus bekerja di INTIMEDIA, software development company, sebagai *Solution Architect Specialist* dan juga salah satu staff pengajar di jurusan teknik informatika - Universitas Bina Nusantara di Jakarta. Agus Kurniawan juga adalah salah satu leader pada komunitas C# Indonesia yang merupakan bagian dari INDC dan memperoleh award MVP untuk C# dari Microsoft Redmond.

Kupersembahkan buku ini : Ibu dan Bapak yang ada di Tasikmalaya

Agus Kurniawan telah memberikan kontribusi di buku ini pada bab 15



Risman Adnan

Risman Adnan adalah salah satu pendiri dari INDC (Indonesia .NET Developer Community) yang saat ini bekerja sebagai *Developer Evangelist* di Microsoft Indonesia. Sebelum berkecimpung di dunia IT, Risman Adnan adalah scientific programmer yang bekerja dalam penelitian mengenai semikonduktor amorf silikon karbon. OOP, .NET, C#

Kupersembahkan buku ini : "To Principessa..."

Risman Adnan telah memberikan kontribusi di buku ini pada bab 12 dan 14



Panji Aryaputra

Pemrograman adalah sesuatu yang dikenal Panji sejak tahun 1992, bahkan sebelum mengenal OS dan hardware. Saat awal belajar Pascal, ia pun lebih tahu sintak bahasa dibanding cara mem-boot komputer. Sejak mempunyai PC 286, ia makin giat menekuni beragam aplikasi pemrograman, mulai dari Pascal, C/C++, Delphi, VB, Java, hingga bahasa terbaru .Net. Selain pemrograman dan OS, ia sempat "terdampar" di bidang automation dengan tool-tool seperti Wonderware Intouch, Intellution IFix, dan PCVue32. Di samping menekuni buku-buku StarTrek, Panji mengisi waktu senggang dengan aktif dalam berbagai mailing list pemrograman di Indonesia.

Kupersembahkan buku ini : Saya mengucapkan terima kasih pada teman-teman yang selama ini telah bermurah hati meminjamkan komputer dan/atau mengajari saya. Semoga saya bisa memberi sebanyak saya menerima.

Panji Aryaputra telah memberikan kontribusi di buku ini pada bab 4 dan 6



Norman Sasono

Norman Sasono adalah seorang Microsoft MVP untuk spesialisasi Visual C#. Ia memegang sertifikasi MCSD.NET dan saat ini bekerja sebagai Senior Developer di Intimedia, sebuah Enterprise IT Solution Provider. Pekerjaan sehari-harinya adalah membuat arsitektur dan desain suatu aplikasi dan memimpin sebuah development team untuk membangun aplikasi tersebut. Ia juga memberi konsultasi rekomendasi arsitektur dan desain aplikasi. Telah menyelesaikan beberapa proyek berbasis .NET, termasuk sebuah proyek dengan tim dari Microsoft. Ia juga memberi berbagai training seputar .NET dan sering menjadi pembicara di berbagai event/seminar Microsoft. Aktif sebagai salah satu group leader di komunitas .NET developer di Indonesia (INDC), khususnya C#. Alumni ITS ini tertarik pada topik-topik tentang *Application Architecture, Distributed Systems, Design Patterns, Refactoring* dan *Agile Development*.

Kupersembahkan buku ini : To Ovie & Cristo

Norman Sasono telah memberikan kontribusi di buku ini pada bab 7 dan 9



Ali Ahmad Heryana

Ali Ahmad Heryana, saat ini bekerja sebagai Developer di PT Ericsson Indonesia. Lahir di Subang - Jawa Barat dan menamatkan kuliahnya di salah satu PT di Bandung. Ali adalah salah seorang pencetus munculnya web csharpindonesia.net dan aspxindonesia.net yang sekarang menjadi bagian dari INDC. Semua hal yang berhubungan dengan IT dan Telekomunikasi sangat digemari untuk dipelajarinya. Selain sebagai salah satu group leader di INDC pada komunitas ASP.NET Indonesia, di waktu luangnya, Ali tidak ketinggalan untuk mengikuti dunia otomotif terutama yang berhubungan dengan Motor.

Kupersembahkan buku ini:

" buku/ebook ini dipersembahkan buat putri pertama ku Nabila & istriku Azan N"

Ali Ahmad Heryana telah memberikan kontribusi di buku ini pada bab 1 dan 2



M. Fathur Rahman

M. Fathur Rahman adalah Developer disatu perusahaan IT di Indonesia. Lahir di Boyalali pada 11 Januari 1976. Fatur lulusan dari Teknik Nuklir UGM. Disela waktu luangnya Fatur sering memainkan musik klasik dengan gitar terutama karya S bach.

Kupersembahkan buku ini:

"Saya menulis bagian ini untuk enggkau,

Yang tidak sombong.

Yang mau menghargai orang lain.

Yang mau berbagi.

Yang mau bergandeng tangan.

Yang mau membela yang benar dengan cara yang benar.

Yang mau menegakkan keadilan.

Yang mau menasihati dengan kebenaran.

Yang mau bertindak dengan kesabaran.

Tidak untuk enggkau yang sebaliknya."

M. Fathur Rahman telah memberikan kontribusi di buku ini pada bab 8 dan 10



I Wayan Saryada

I Wayan Saryada adalah salah satu Developer di perusahaan IT di Bali. Lahir di Denpasar, 30 Desember 1976. Saryada lulusan dari Manajemen Informatika di STIKOM Surabaya.

Kupersembahkan buku ini: keluarga tercinta: Untuk komunitas C# Indonesia

I Wayan Saryada telah memberikan kontribusi di buku ini pada bab 3 dan 11



Adi Wirasta

Adi Wirasta adalah salah satu Developer di PT. Internet Cipta Rekayasa. Lahir di Jakarta, 12 Januari 1980. Adi lulusan dari Ilmu Komputer di IPB. Teknologi yang digeluti sampai sekarang ini adalah BioInformatika (www.bioinformasi.com) .

Kupersembahkan buku ini: keluarga tercinta: Ibu, Lisa, Dewi, Alex dan Ario

Adi Wirasta telah memberikan kontribusi di buku ini pada bab 5 dan 13

Daftar Isi

Project Otak	3
Credits	3
Kata Pengantar	4
Tentang Penulis	5
Daftar Isi	9
1. Pengenalan Framework .NET	14
1.1 Apakah Itu Framework .NET ?	14
1.2 Arsitektur Framework .NET	15
1.3 Keuntungan Framework .NET	16
2. Pengenalan Bahasa C#	17
2.1 Mengapa C# ?	17
2.2 Pemilihan Editor	19
2.2.1 Notepad	19
2.2.2 Visual Studio 6	20
2.2.2 Visual Studio .NET	20
2.2.3 Editor Lainnya	21
2.3 Hello World	21
2.5 Struktur Penulisan Kode C#	22
2.5.1 Bagian utama struktur penulisan kode C#	22
2.5.2 Aksesories penulisan kode C#	23
2.6 Kompilasi (<i>Compile</i>) Aplikasi	24
2.7 Menjalankan Aplikasi	26
3. Variabel dan Ekspresi	28
3.1 Variabel	28
3.1.1 Nama Variabel	28
3.1.2 Kategori Variabel	29
3.1.2.1 Parameters	30
3.1.3 Mendeklarasikan Variabel	31
3.1.4 Assigment / Pemberian Nilai	32
3.1.5 Default Value	33
3.1.6 Tipe Data Numerik	33
3.1.7 Contoh Program	34
3.2 Ekspresi	35
3.2.1 Checked vs Unchecked	37
4. Flow Control	39
4.1 Kondisi dan Aksi	39
4.2 Selection Statement	40
4.2.1 if	40
4.2.2 if-else	41

4.2.3 switch	41
4.3 Iteration Statement	43
4.3.1 while.....	43
4.3.2 do.....	44
4.3.3 for.....	44
4.3.4 foreach.....	45
4.4 Jump Statement.....	45
4.4.1 break.....	46
4.4.2 continue.....	46
4.4.3 goto	47
4.4.4 return	47
4.4.5 throw	48
5. methods	50
5.1 Pengenalan Method.....	50
5.2 Implementasi methods	51
5.2.1 static methods.....	51
5.2.2 non-static method.....	52
5.3. Modifier	54
6. Debugging dan Penanganan Error	59
6.1 Penanganan Error.....	59
6.2 Exception	60
6.3 Try statement	60
6.4 Multiple Catch Block.....	62
6.5 User-Defined/Custom Exception	63
6.6 Kapan Menggunakan Try Statement	63
6.7 Debugging.....	64
6.7.1 Breakpoint.....	65
6.7.2 Autos, Locals, dan Watch	67
6.7.3 Step Into, Step Over, dan Step Out	68
6.7.4 Call Stack.....	71
6.7.5 Command Window	72
6.7.6 Output Window.....	73
7. Pemrograman Object-Oriented	74
7.1 Object.....	74
7.1.1 Apa itu Object?	74
7.1.2 Definisi Formal	75
7.1.3 State.....	75
7.1.4 Behavior.....	76
7.1.5 Identitas.....	77
7.2 Prinsip Dasar Object-Oriented.....	77
7.2.1 Abstraction.....	77
7.2.2 Encapsulation.....	78
7.2.3 Modularity.....	78
7.2.4 Hirarki	79
7.3 Object dan Class	79
7.3.1 Class.....	79

7.3.2 Attribute	80
7.3.3 Operation.....	81
7.3.4 Polymorphism	81
7.3.5 Inheritance.....	82
7.4 Object-Oriented C#.....	84
8. Class	89
8.1 Deklarasi Class.....	89
8.2. Contoh Implementasi Class	90
8.3 Class untuk Object yang Tidak Berubah	93
8.4. Class Abstrak	94
8.5 Inheritance.....	98
8.6 Lebih Jauh dengan Class.....	98
8.6.1 Class Abstrak	99
8.6.2. Sealed Class	99
8.6.3. Class Member	100
8.6.3.1. Konstanta	100
8.6.3.2. Field	102
8.6.3.3 Method	103
Value Parameter.....	104
Reference Parameter	105
Output parameter	106
Parameter Array	107
Virtual Method.....	108
Override Method.....	110
Abstract Method.....	110
Method Overloading	111
Constructor Overloading.....	112
9. Inheritance	113
9.1 Class dan Inheritance	113
9.1.1 Base Class	113
9.1.2 Base Class Member.....	114
9.1.3 Base Class Constructor	115
9.2 Implementasi Method	117
9.2.1 Virtual Method.....	117
9.2.2 Override Method.....	118
9.2.3 Keyword new	119
9.3 Sealed Class	119
9.4 Abstract Class	120
9.4.1 Abstract Class	120
9.4.2 Abstract Method.....	120
9.5 Penutup	122
10. Interface	123
10.1 Deklarasi Interface	123
10.2 Implementasi Interface.....	124
11. Collection	130
11.1 Array	130

11.1.1 Mendeklarasikan Array	130
11.1.2 Meng-Initialize Array	131
11.1.3 Mengakses Array	131
11.1.4 Contoh Program	131
11.2 Collection	132
11.2.1 ICollection	132
11.2.2 IEnumerable	133
11.2.3 IList	133
11.2.4 IDictionary	133
11.2.5 Queue	134
11.2.6 Stack	135
11.2.7 ArrayList	136
11.2.8 StringCollection	136
11.2.9 Hashtable	137
11.2.10 StringDictionary	138
12. Namespace	139
12.1. Definisi Namespace	139
12.2. Namespace dan Using	139
12.3. Namespace Bertingkat	141
12.4. Namespace dan Assembly	142
13. File	144
13.1 Pengenalan Namespace System.IO	144
13.2 Implementasi System.IO	146
13.2.1 Implementasi Class Path	146
13.2.2 Implementasi beberapa class System.IO	146
14. Delegate dan Event Handler	148
14.1. Sekilas tentang Delegate	148
14.2. Delegate untuk Callback	148
14.2.1 Proses Asynchronous	151
14.2.2 Memasukkan Kode Tambahan ke Kode Suatu Class	151
14.3 Delegate Adalah Class	151
14.4 Mendefinisikan Delegate Sebagai Static Member	154
14.5 Property dan Delegate	155
14.6 Multicast Delegate	157
14.7. Mendefinisikan Event Dari Multicast Delegate	160
15. Studi Kasus – Penerapan OOP Dengan C#	163
15.1 Pendahuluan	163
15.2 Membangun Aplikasi Perhitungan Gaji	163
15.3 Design Aplikasi	163
15.4 Diagram Use Cases	163
15.5 Diagram Class	164
15.5.1 Menggunakan Inheritance	164
15.5.2 Menggunakan Interface	165
15.6 Implementasi Diagram Class	166
15.6.1 Menggunakan Konsep Inheritance	166
15.6.2 Menggunakan Konsep Interface	167

15.7 Menjalankan Aplikasi	168
15.7.1 Konsep Inheritance	168
15.7.2 Konsep Interface	169
Daftar Pustaka	171
Lampiran	172

1. Pengenalan Framework .NET

Ali Ahmad H

Untuk dapat lebih memahami tentang bahasa C# dan dapat memanfaatkannya secara lebih maksimal, maka sebelum kita memulai membahas tentang apa itu bahasa C#, ada baiknya kita mempelajari terlebih dahulu tentang Microsoft Framework .NET yang merupakan komponen yang tidak bisa dipisahkan dari bahasa C# itu sendiri.

Adapun penjelasan pada bab ini akan diawali dengan penjelasan tentang istilah Framework .NET istilah-istilah lain yang berhubungan dengannya.

1.1 Apakah Itu Framework .NET ?

Framework .NET adalah suatu komponen windows yang terintegrasi yang dibuat dengan tujuan untuk mensupport pengembangan berbagai macam jenis aplikasi serta untuk dapat menjalankan berbagai macam aplikasi generasi mendatang termasuk pengembangan aplikasi Web Services XML.

Framework .NET di design untuk dapat memenuhi beberapa tujuan berikut ini :

- Untuk menyediakan environment kerja yang konsisten bagi bahasa pemrograman yang berorientasi objek (*object-oriented programming - OOP*) baik kode objek itu di simpan dan di eksekusi secara lokal, atau dieksekusi secara lokal tapi didistribusikan melalui internet atau di eksekusi secara remote.
- Untuk menyediakan environment kerja di dalam mengeksekusi kode yang dapat meminimaliasi proses software *deployment* dan menghindari konflik penggunaan versi software yang di buat.
- Untuk menyediakan environment kerja yang aman dalam hal pengeksekusian kode, termasuk kode yang dibuat oleh pihak ketiga (*third party*).
- Untuk menyediakan environment kerja yang dapat mengurangi masalah pada persoalan performa dari kode atau dari lingkungan *interpreter* nya.
- Membuat para developer lebih mudah mengembangkan berbagai macam jenis aplikasi yang lebih bervariasi, seperti aplikasi berbasis windows dan aplikasi berbasis web.
- Membangun semua komunikasi yang ada di dalam standar industri untuk memastikan bahwa semua kode aplikasi yang berbasis Framework .NET dapat berintegrasi dengan berbagai macam kode aplikasi lain.

Sebagai salah satu sarana untuk dapat memenuhi tujuan di atas, maka dibuatlah berbagai macam bahasa pemrograman yang dapat digunakan dan dapat berjalan di atas platform Framework .NET seperti bahasa C#, VB.NET, J#, Perl.NET dan lain-lain. Masing-masing bahasa tersebut mempunyai kelebihan dan kekurangannya masing-masing, namun yang pasti, apapun bahasa pemrograman yang digunakan, semuanya

akan dapat saling berkomunikasi dan saling compatible satu dengan yang lainnya dengan bantuan Framework .NET

1.2 Arsitektur Framework .NET

Framework .NET terdiri dari dua buah komponen utama, yaitu *Common Language Runtime (CLR)* dan *.NET Framework Class Library* atau kadang juga sering disebut dengan *Base Class Library (BCL)*.

Common Language Runtime (CLR) adalah pondasi utama dari Framework .NET. CLR merupakan komponen yang bertanggung jawab terhadap berbagai macam hal, seperti bertanggung jawab untuk melakukan manajemen memory, melakukan eksekusi kode, melakukan verifikasi terhadap keamanan kode, menentukan hak akses dari kode, melakukan kompilasi kode, dan berbagai layanan system lainnya. Dengan adanya fungsi CLR ini, maka aplikasi berbasis .NET biasa juga disebut dengan *managed code*, sedangkan aplikasi di luar itu biasa disebut dengan *un-managed code*.

Berikut ini beberapa hal yang disediakan CLR bagi para developer:

- Dapat lebih menyederhakan proses pengembangan aplikasi.
- Memungkinkan adanya variasi dan integrasi dari berbagai bahasa pemrograman yang ada di lingkungan Framework .NET
- Keamanan dengan melakukan indenting pada kode aplikasi.
- Bersifat *Assembly* pada saat proses deployment / kompilasi
- Melakukan versioning sebuah komponen yang bisa di daur ulang.
- Memungkinkan penggunaan kembali kode, dengan adanya sifat inheritance.
- Melakukan pengaturan / manajemen tentang *lifetime* sebuah objek.
- Melakukan penganalisaan objek-objek secara otomatis.

CLR akan melakukan kompilasi kode-kode aplikasi kita menjadi bahasa assembly MSIL (Microsoft Intermediate Language). Proses kompilasi ini sendiri dilakukan oleh komponen yang bernama *Just In Time (JIT)*. JIT hanya akan mengkompilasi metode-metode yang memang digunakan dalam aplikasi, dan hasil kompilasi ini sendiri di *chace* di dalam mesin dan akan dikompilasi kembali jika memang ada perubahan pada kode aplikasi kita.

.NET Framework Class Library atau sering juga disebut **Base Case Library (BCL)** adalah koleksi dari *reusable types* yang sangat terintegrasi secara melekat dengan CLR. *Class library* bersifat berorientasi terhadap objek yang akan menyediakan *types* dari fungsi-fungsi *managed code*. Hal ini tidak hanya berpengaruh kepada kemudahan dalam hal penggunaan, tetapi juga dapat mengurangi waktu yang diperlukan pada saat eksekusi. Dengan sifat tersebut, maka komponen pihak ketiga akan dengan mudah diaplikasikan ke dalam aplikasi yang dibuat.

Dengan adanya BCL ini, maka kita bisa menggunakan Framework .NET untuk membuat berbagai macam aplikasi, seperti :

- Aplikasi *console*
- Aplikasi berbasis windowd (*Windows Form*)
- Aplikasi ASP.NET (berbasis web)

- Aplikasi Web Services XML
- Aplikasi berbasis *Windows Services*

Jika kita membuat sekumpulan Class untuk membuat aplikasi berbasis windows, maka Class-Class itu bisa kita gunakan untuk jenis aplikasi lain, seperti aplikasi berbasis web (ASP.NET).

1.3 Keuntungan Framework .NET

Berikut beberapa keuntungan dari Framework .NET

Mudah

Kemudahan di sini lebih ke arah pada kemudahan bagi para developer untuk membuat aplikasi yang dijalankan pada lingkungan Framework .NET. Beberapa hal yang merepotkan developer pada saat membuat aplikasi, telah di hilangkan atau di ambil alih kemampuannya oleh Framework .NET, misalnya masalah *lifetime* sebuah objek yang biasanya luput dari perhatian developer pada saat proses pembuatan aplikasi. Masalah ini telah ditangani dan diatur secara otomatis oleh Framework .NET melalui komponen yang bernama *Garbage Collector* yang bertanggung jawab untuk mencari dan membuang objek yang sudah tidak terpakai secara otomatis.

Efisien

Kemudahan pada saat proses pembuatan aplikasi, akan berimplikasi terhadap efisiensi dari suatu proses produktivitas, baik efisien dalam hal waktu pembuatan aplikasi atau juga efisien dalam hal lain, seperti biaya (*cost*).

Konsisten

Kemudahan-kemudahan pada saat proses pembuatan aplikasi, juga bisa berimplikasi terhadap konsistensi pada aplikasi yang kita buat. Misalnya, dengan adanya BCL, maka kita bisa menggunakan objek atau Class yang dibuat untuk aplikasi berbasis windows pada aplikasi berbasis web. Dengan adanya kode yang bisa dintegrasikan ke dalam berbagai macam aplikasi ini, maka konsistensi kode-kode aplikasi kita dapat terjaga.

Produktivitas

Semua kemudahan-kemudahan di atas, pada akhirnya akan membuat produktivitas menjadi lebih baik. Produktivitas naik, terutama produktivitas para developer, akan berdampak pada meningkatnya produktivitas suatu perusahaan.

2. Pengenalan Bahasa C#

Ali Ahmad H

C# (dibaca "See-Sharp") adalah bahasa pemrograman baru yang diciptakan oleh Microsoft (dikembangkan dibawah kepemimpinan Anders Hejlsberg yang notabene juga telah menciptakan berbagai macam bahasa pemrograman termasuk Borland Turbo C++ dan Borland Delphi). Bahasa C# juga telah di standarisasi secara internasional oleh ECMA.

Seperti halnya bahasa pemrograman yang lain, C# bisa digunakan untuk membangun berbagai macam jenis aplikasi, seperti aplikasi berbasis windows (desktop) dan aplikasi berbasis web serta aplikasi berbasis web services.

2.1 Mengapa C# ?

Pertanyaan di atas mungkin muncul di benak kita semua pada saat pertama kali mendengar tentang C#. Hal tersebut sangat beralasan, dengan melihat kenyataan bahwa sebelum C# muncul, telah banyak bahasa pemrograman yang ada, seperti C, C++, Java, Perl dan lain-lain.

Ada beberapa alasan kenapa memilih C#, yaitu :

Sederhana (simple)

C# menghilangkan beberapa hal yang bersifat kompleks yang terdapat dalam beberapa macam bahasa pemrograman seperti Java dan C++, termasuk diantaranya mengilangkan *macro*, *templates*, *multiple inheritance* dan *virtual base classes*. Hal-hal tersebut yang dapat menyebabkan kebingungan pada saat menggunakannya, dan juga berpotensi dapat menjadi masalah bagi para programmer C++. Jika anda pertama kali belajar bahasa C# sebagai bahasa pemrograman, maka hal-hal tersebut di atas tidak akan membuat waktu anda terbuang terlalu banyak untuk mempelajarinya.

C# bersifat sederhana, karena bahasa ini didasarkan kepada bahasa C dan C++. Jika anda familiar dengan C dan C++ atau bahkan Java, anda akan menemukan aspek-aspek yang begitu familiar, seperti *statements*, *expression*, *operators*, dan beberapa fungsi yang diadopsi langsung dari C dan C++, tetapi dengan berbagai perbaikan yang membuat bahasanya menjadi lebih sederhana.

Modern

Apa yang membuat C# menjadi suatu bahasa pemrograman yang modern? Jawabannya adalah adanya beberapa fitur seperti *exception handling*, *garbage collection*, *extensible data types*, dan *code security* (keamanan kode/bahasa pemrograman). Dengan adanya fitur-fitur tersebut, menjadikan bahasa C# sebagai bahasa pemrograman yang modern.

Object-Oriented Language

Kunci dari bahasa pemrograman yang bersifat *Object Oriented* adalah *encapsulation*, *inheritance*, dan *polymorphism*. Secara sederhana, istilah-istilah tersebut bisa didefinisikan sebagai berikut (definisi dan penjelasan lebih lanjut akan di uraikan pada bab-bab selanjutnya).

encapsulation, dimana semua fungsi ditempatkan dalam satu paket (*single package*).
inheritance, adalah suatu cara yang terstruktur dari suatu kode-kode pemrograman dan fungsi untuk menjadi sebuah program baru dan berbentuk suatu paket.
polymorphism, adalah kemampuan untuk mengadaptasi apa yang diperlukan untuk dikerjakan.

Sifat-sifat tersebut di atas, telah di miliki oleh C# sehingga bahasa C# merupakan bahasa yang bersifat *Object Oriented*.

Powerfull dan fleksibel

C# bisa digunakan untuk membuat berbagai macam aplikasi, seperti aplikasi pengolah kata, grafik, spreadsheets, atau bahkan membuat kompilerv untuk sebuah bahasa pemrograman.

Efisien

C# adalah bahasa pemrograman yang menggunakan jumlah kata-kata yang tidak terlalu banyak. C# hanya berisi kata-kata yang biasa disebut dengan *keywords*. *Keywords* ini digunakan untuk menjelaskan berbagai macam informasi. Jika anda berpikiran bahwa bahasa pemrograman yang menggunakan sangat banyak kata-kata (*keywords*) akan lebih powerfull, maka jawabannya adalah “pemikiran itu tidak selalu benar”, karena hal itu justru bisa menambah kerumitan para developer pada saat membuat suatu aplikasi. Berikut daftar *keywords* yang ada dalam bahasa C#:

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	
goto	if	implicit	in	int
interface	internal	is	lock	long
namespace	new	null	object	operator
out	override	params	private	protected
public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	
static	string	struct	switch	this
throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using
virtual	void	while		

Table 2.1 Daftar keywords pada bahasa C#

Modular

Kode C# ditulis dengan pembagian masing Class-Class (*classes*) yang terdiri dari beberapa *routines* yang disebut sebagai *member methods*. Class-Class dan metode-metode ini dapat digunakan kembali oleh program atau aplikasi lain. Hanya dengan memberikan informasi yang dibutuhkan oleh Class dan metode yang dimaksud, maka kita akan dapat membuat suata kode yang dapat digunakan oleh satu atau beberapa aplikasi dan program (*reusable code*)

C# akan menjadi populer

Dengan dukungan penuh dari Microsoft yang akan mengeluarkan produk-produk utamanya dengan dukungan Framework .NET, maka masa depan bahasa C# sebagai salah satu bahasa pemrograman yang ada di dalam lingkungan Framework .NET akan lebih baik.

2.2 Pemilihan Editor

Sebelum kita memulai menulis kode pemrograman dengan C#, kita perlu mengetahui dan memilih editor yang akan kita gunakan. Berikut ini ada beberapa editor yang bisa digunakan, dimana editor-editor tersebut mempunyai kelebihan dan kekurangan masing-masing.

2.2.1 Notepad

Microsoft Notepad telah banyak digunakan sebagai editor berbasis teks untuk menulis berbagai macam bahasa pemrograman, termasuk C#. Namun dengan beberapa alasan berikut, Notepad tidak di rekomendasikan untuk digunakan sebagai editor untuk membuat aplikasi dengan C#:

File-file C# disimpan dengan ekstension *.cs*, jika kita tidak hati-hati pada saat menyimpan file C# di Notepad, misal kita bermaksud menyimpan file dengan nama *test.cs* maka tidak tertutup kemungkinan file tersebut akan menjadi *test.cs.txt* kecuali kita telah menseeting terlebih dahulu box drop down list pada fungsi *Save As* menjadi "All Files".

Notepad tidak dapat menampilkan nomor baris, hal ini akan menyulitkan kita pada saat kita mendebug dari kemungkinan error yang ada yang memberikan kita informasi di baris ke berapa error tersebut terjadi.

Notepad tidak dapat melakukan *automatic indenting* (tab secara otomatis), sehingga kita harus melakukannya secara manual, hal ini jelas sangat merepotkan terutama jika kode yang kita buat telah banyak dan kompleks.

Beberapa permasalahan di atas bisa dijadikan sebagai alasan, kenapa penggunaan Notepad tidak direkomendasikan sebagai editor C#, walaupun untuk membuat aplikasi-aplikasi sederhana dan kecil, Notepad masih bisa digunakan sebagai editor dengan tetap memperhatikan beberapa permasalahan tersebut di atas.

2.2.2 Visual Studio 6

Jika anda telah terbiasa menggunakan Visual Studio 6, maka tools tersebut bisa digunakan untuk membuat aplikasi dengan C#, khususnya dengan menggunakan editor Microsoft Visual C++.

Salah satu keuntungan menggunakan editor khusus buat pemrograman (seperti Microsoft Visual C++) adalah adanya *syntax highlighting*, yang memudahkan kita pada saat membaca dan menganalisa kode-kode program kita. Namun, karena Visual Studio 6 (khususnya Visual C++) ini di buat sebelum adanya bahasa C#, maka perlu sedikit "kreatifitas" kita untuk memodifikasi setting editor tersebut agar dapat menampilkan *syntax highlighting C#*.

Trik yang bisa dilakukan adalah dengan mengedit *registry key* untuk Visual Studio 6 dengan menggunakan Regedit.exe atau editor registry lainnya, edit pada bagian berikut:

```
HKEY_CURRENT_USER\Software\Microsoft\DevStudio\6.0\
Text Editor\Tabs\Language Settings\C\C++\FileExtensions
```

Pada bagian *value data*, akan mempunyai nilai berupa daftar ekstension yang akan dikenali secara default oleh tools, seperti di bawah ini:

```
cpp; cxx; c; h; hxx; hpp; i; nl; tlh; tli; rc; rc2
```

tambahkan ekstension .cs (tanpa tanda titik) pada bagian value data sehingga nilainya akan menjadi seperti berikut ini:

```
cpp; cxx; c; h; hxx; hpp; i; nl; tlh; tli; rc; rc2; cs
```

Sekarang, jika kita membuka file yang berekstension .cs dengan menggunakan Microsoft Visual C++, maka tools tersebut akan mengenali file tersebut sebagai salah satu jenis file yang di support secara default.

Selanjutnya, kita perlu untuk mendeklarasikan kata-kata kunci C# pada Visual Studio kita, yang bisa dilakukan dengan mengedit file *usertype.dat* yang biasanya di simpan satu direktori dengan file *msdev.exe*. Untuk melihat perubahannya, maka kita perlu merestart Visual Studio kita terlebih dahulu.

2.2.2 Visual Studio .NET

Visual Studio .NET merupakan editor yang paling ideal untuk membuat aplikasi yang berbasis Framework .NET, termasuk aplikasi dengan bahasa C#. Di dalam buku ini, semua kode program akan di tulis dengan menggunakan Visual Studio .NET 2003. Dengan editor ini, maka kita akan bisa memanfaatkan kemampuan C# secara maksimal. Editor ini tidak hanya menyediakan berbagai macam tools dan wizard untuk membuat aplikasi C#, tapi juga termasuk fitur-fitur produktif seperti *IntelliSense* dan bantuan yang dinamis.

Dengan *IntelliSense*, jika kita mengetikan nama sebuah *namespace* atau nama Class, maka anggota dari *namespace* atau Class itu akan secara otomatis di munculkan sehingga kita tidak usah mengingat anggota dari semua I atau semua Class yang kita gunakan. *IntelliSense*, juga akan menampilkan semua argumen dan jenis typenya ketika

kita mengetikkan nama dari sebuah metode. Visual Studio 6 juga telah memiliki kemampuan ini, hanya saja Visual Studio 6 tidak mensupport jenis type dan Class-Class yang termasuk di dalam lingkungan Framework .NET.

Fitur bantuan yang dinamis (*dynamic help*) merupakan fitur yang baru yang ada di dalam “keluarga” editor Visual Studio. Ketika kita mengetikkan sebuah code pada editor, sebuah jendela yang terpisah akan menampilkan topik-topik bantuan dan penjelasan yang berhubungan dengan kata-kata tempat di mana kita menempatkan cursor. Misalnya, ketika kita mengetikkan kata *namespace*, maka di jendela yang terpisah tadi akan dimunculkan topik-topik yang berhubungan dengan kata kunci *namespace*. Hal ini tentu akan sangat membantu kita pada saat pembuatan program untuk dapat lebih memahami dan mengerti tentang kata-kata / kode yang kita tulis.

2.2.3 Editor Lainnya

Selain editor-editor yang telah disebutkan di atas, masih banyak beberapa editor lain yang bisa digunakan untuk membuat aplikasi dengan C#, seperti Visual SlickEdit dari MicroEdge, WebMatrikx untuk aplikasi C# berbasis web, editor text seperti UltraEdit, Macromedia Homesite, dll. Editor-editor tersebut tidak akan dibahas di buku ini, namun intinya, kita bisa menggunakan editor-editor tersebut dengan kelebihan dan kekurangannya masing-masing.

2.3 Hello World

Asumsikan bahwa kita telah memilih editor untuk membuat aplikasi C#. Selanjutnya, kita akan mencoba membuat aplikasi C# yang sederhana. Untuk pertama kali, kita akan mencoba membuat aplikasi Hello World sederhana dengan menggunakan Notepad. Buka notepad dan tulis kode dibawah ini, kemudian simpan dengan nama helloworld.cs:

```
class HelloWorld
{
    // Bagian utama program C#
    public static void Main()
    {
        System.Console.WriteLine("Hello, World");
    }
}
```

Pada bagian ini, kita tidak akan membahas secara rinci baris per baris dari kode program kita di atas, di sini hanya akan diperlihatkan bagaimana struktur penulisan aplikasi C#, melakukan kompilasi, dan kemudian menjalankannya. Pembahasan yang lebih rinci tentang syntax pemrograman, arti masing-masing kata kunci (*keywords*) akan dibahas pada bagian selanjutnya.

2.5 Struktur Penulisan Kode C#

2.5.1 Bagian utama struktur penulisan kode C#

Program helloworld.cs di atas merupakan struktur kode program C# yang paling sederhana.

Kode program diawali dengan mendeklarasikan nama Class atau *namespace* (penjelasan yang lebih rinci tentang Class *namespace*, akan di bahas pada bab selanjutnya).

```
class HelloWorld
```

Kemudian seluruh aplikasi dibuka dengan tanda "{" dan pada akhir kode ditutup dengan tanda "}".

```
class HelloWorld  
{  
}
```

Aplikasi C# dibangun oleh satu atau beberapa fungsi yang diletakan di dalam sebuah Class. Nama suatu fungsi pada C# harus diawali dengan huruf, atau garis bawah "_" yang kemudian bisa diikuti oleh huruf, angka atau garis bawah. Pada bagian akhir nama fungsi digunakan tanda kurung buka dan kurung tutup "()". Penamaan fungsi tidak boleh mengandung spasi. Awal dan akhir suatu fungsi di mulai dengan tanda "{" dan diakhiri dengan tanda "}". Berikut contoh penamaan fungsi yang diletakan di dalam Class:

```
class HelloWorld  
{  
    NamaFungsi 1()  
}  
}
```

atau

```
class HelloWorld  
{  
    _NamaFungsi 1()  
}  
}
```

Nama fungsi utama yang biasa digunakan pada aplikasi C# adalah Main. Setiap fungsi memiliki sifat fungsi, seperti public dan static. Selain itu, untuk menandakan apakah itu blok fungsi atau bukan, sebelum nama fungsi digunakan void, berikut contoh lengkapnya:

```
class HelloWorld  
{  
    public static void Main()  
    {  
    }  
}
```

Penjelasan tentang `public`, `static` dan `void` itu sendiri akan dibahas lebih rinci lagi di bab selanjutnya.

Di dalam sebuah fungsi, berisikan sekumpulan perintah-perintah, dimana perintah satu dengan lainnya akan dipisahkan atau diakhiri dengan tanda “;”. Pada contoh `helloworld.cs` sebelumnya, perintah yang digunakan adalah untuk mengeluarkan output berupa tulisan “Hello, World” yang akan tampil pada mode console (mode dos prompt), seperti kode berikut ini:

```
class HelloWorld
{
    // Bagian utama program C#
    public static void Main()
    {
        System.Console.WriteLine("Hello, World");
    }
}
```

Penjelasan tentang `System`, `Console`, `WriteLine` dan kata-kata kunci lainnya akan dibahas di bab selanjutnya. Pada bagian ini, Anda hanya dikenalkan pada tata cara struktur penulisan kode program pada aplikasi C#.

2.5.2 Aksesories penulisan kode C#

Komentar

Ada 2 cara yang bisa digunakan untuk menulis komentar di C#. Untuk komentar satu baris atau perbaris, bisa digunakan tanda “//”, semua yang ditulis setelah tanda ini, dianggap sebagai komentar yang tidak akan di eksekusi. Perpindahan baris komentar satu dengan yang lainnya dipisahkan dengan “*enter*”, contohnya:

```
// Ini adalah baris komentar.

// Baris komentar 1
// Baris komentar 2
```

Cara lain adalah dengan diawali tanda “/*” dan diakhiri dengan tanda “*/”. Tanda ini biasanya digunakan untuk sebuah komentar yang panjang, seperti contoh berikut :

```
/* Ini adalah baris komentar
   Apapun yang di tulis di sini tidak akan di eksekusi
```

Escape Sequences

Escape Sequences adalah karakter-karakter khusus yang tidak akan ditampilkan. Contohnya, ada karakter yang digunakan sebagai tanda akhir dari suatu baris yang memerintahkan program untuk melanjutkan ke baris berikutnya. Cara penulisannya diawali dengan tanda \ diikuti dengan karakter khusus (dalam contoh ini adalah “n”) sehingga penulisannya menjadi \n.

Berikut beberapa karakter khusus pada C#:

Escape Sequence	Nama	ASCII	Keterangan
\a	Bell (alert)	007	Menghasilkan suara (beep)
\b	Backspace	008	Mengembalikan posisi kursor ke sebelumnya
\t	Horizontal Tab	009	Meletakkan posisi kursor di pemberhentian Tab berikutnya
\n	New line	010	Meletakkan posisi kursor pada baris baru
\v	Vertical Tab	011	Melakukan Tab secara vertical
\f	Form feed	012	
\r	Carriage return	013	Memberikan nilai enter
\"	Double Quote	034	Menampilkan <i>double quote</i> (")
\'	Apostrophe	039	Menamppilkan <i>apostrophe</i> (')
\?	Question mark	063	Menampilkan tanda Tanya
\\	Backslash	092	Menampilkan <i>backslash</i> (\)
\0	Null	000	Menampilkan karakterk Null

Tabel 2.2 Daftar karakter khusus pada bahasa C#

2.6 Kompilasi (*Compile*) Aplikasi

Untuk mengkompilasi aplikasi kita di atas, kita akan menggunakan *command-line compiler*. Cara ini mungkin akan sangat jarang digunakan, terutama jika kita menggunakan Visual Studio .NET untuk membuat aplikasi C# kita. Tujuan penggunaan *command-line compiler* di sini adalah untuk mengenalkan kepada kita semua bahwa proses kompilasi aplikasi C# tidak tergantung kepada tools atau editor yang kita gunakan yang telah mempunyai *compiler built-in*, tapi juga bisa bersifat universal tanpa harus tergantung kepada tools atau editor itu sendiri.

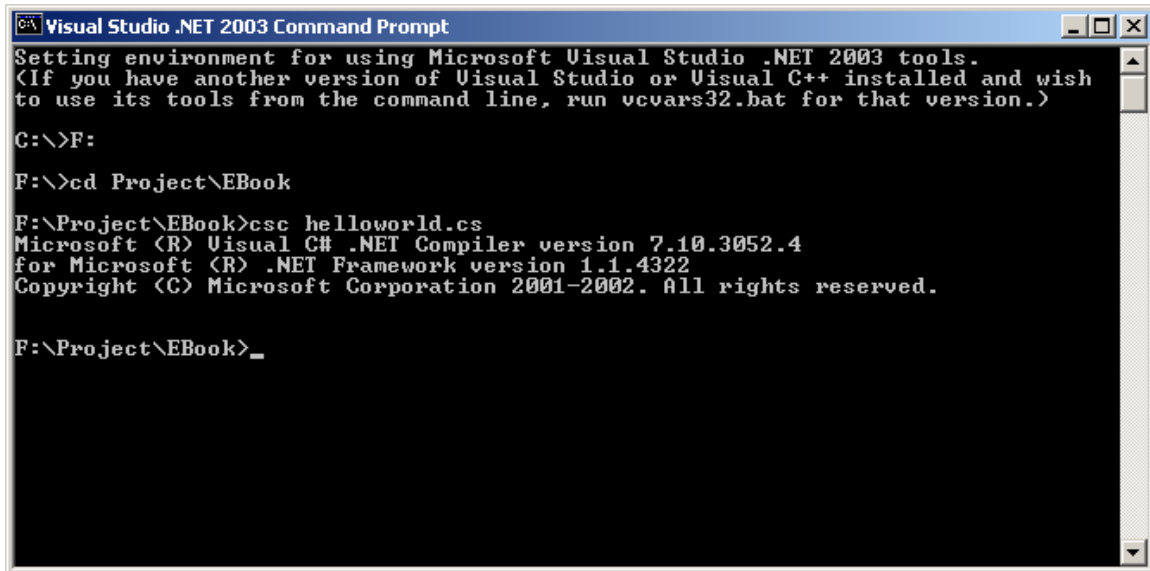
Jika kita telah menginstall Visual Studio .NET (misal versi 2003, yang akan digunakan di buku ini), maka kita bisa menggunakan *command-line tools* dari editor tersebut.

Klik Start -> Program -> Microsoft Visual Studio .NET 2003 -> Visual Studio .NET Tools -> Visual Studio .NET 2003 Command Prompt

Default direktori pada *command line* tersebut adalah C:\ jika aplikasi yang dibuat ada di direktori lain, ubah direktorinya ke tempat di mana kita menyimpan aplikasinya. (Misal di dalam contoh ini ada di direktori F:\Project\Ebook). Kemudian lakukan kompilasi aplikasi kita dengan mengetikan perintah :

```
csc hel loworl d. cs
```

Jika tidak ada error dari proses kompilasi tersebut, maka hasilnya akan kita lihat seperti pada gambar berikut:



```
Visual Studio .NET 2003 Command Prompt
Setting environment for using Microsoft Visual Studio .NET 2003 tools.
<If you have another version of Visual Studio or Visual C++ installed and wish
to use its tools from the command line, run vcvars32.bat for that version.>
C:\>F:
F:\>cd Project\EBook
F:\Project\EBook>csc helloworld.cs
Microsoft (R) Visual C# .NET Compiler version 7.10.3052.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights reserved.
F:\Project\EBook>_
```

Gambar 2-1. Tampilan command-line compiler dari Visual Studio .NET dan hasil kompilasi helloworld.cs tanpa error

Jika kita belum menginstall Visual Studio .NET, maka kita bisa menggunakan command-line yang ada di windows kita.

Klik Start -> Run -> ketik cmd

Selanjutnya, lakukan kompilasi aplikasi kita dengan perintah `csc helloworld.cs`

Jika pada saat melakukan kompilasi ada pesan error berikut :

```
F:\Project\EBook\>csc
'csc' is not recognized as an internal or external command,
operable program or batch file.
```

Hal itu berarti *compiler* `csc.exe` belum dimasukkan ke dalam path windows kita. Untuk dapat melakukan kompilasi, selain kita harus memasukkan `csc.exe` ke dalam path windows, kita juga bisa langsung menjalankan `csc` dari direktori aslinya seperti berikut ini:

```
F:\Project\EBook>C:\WINNT\Microsoft.NET\Framework\v1.1.4322\csc
helloworld.cs
```

Untuk Windows 98 atau XP, ubah direktori `C:\WINNT` menjadi `C:\WINDOWS`.

Angka setelah direktori Framework tergantung dari versi Framework .NET yang kita gunakan, di buku ini digunakan versi v1.1.4322 yaitu Framework .NET 1.1 yang bisa kita install dari CD Visual Studio .NET 2003.

2.7 Menjalankan Aplikasi

Setelah proses kompilasi, kita bisa menjalankan aplikasi helloworld kita dengan mengetikkan perintah berikut :

```
F: \Project\EBook>hel l oworl d  
Hel l o, Worl d
```

Berikut ini kita akan mencoba membuat aplikasi sederhana lain dengan menggunakan Visual Studio .NET 2003.

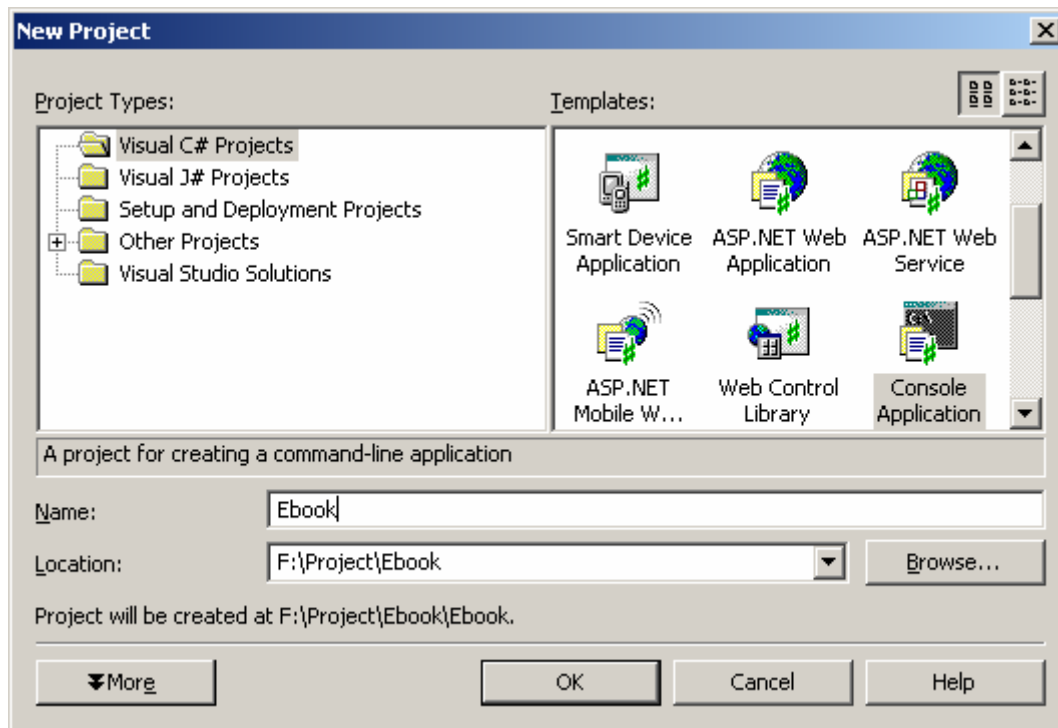
Buka editor Visual Studio .NET 2003 dengan mengklik :

Start -> Program -> Microsoft Visual Studio .NET 2003 -> Visual Studio .NET Tools -> Microsoft Visual Studio .NET 2003

setelah terbuka, pilih:

File -> New -> Project

pada bagian Project Types, pilih Visual C# Projects, dan pada bagian templates, pilih Console Application. Pada box name, ketik nama direktori project, dan di bagian location, masukkan direktori utamanya.



Gambar 2-2. Tampilan Visual Studio .NET 2003 ketika akan membuat project aplikasi C# baru

Visual Studio .NET akan membuat beberapa file secara otomatis. Ketik kode program di bawah ini pada file class1.cs

```
using System;

namespace Ebook
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
            System.Console.WriteLine("Hello, World");
            String str = System.Console.ReadLine();
        }
    }
}
```

Untuk mengkompilasi aplikasi di Visual Studio .NET bias dilakukan dengan meng-klik menu Build -> Build Ebook Project

Hasil output dari project aplikasi console adalah file .exe yang biasanya akan di kompilasi ke direktori /bin/debug

3. Variabel dan Ekspresi

I Wayan Saryada

3.1 Variabel

Sebuah program yang kita buat umumnya merupakan kumpulan data yang akan kita olah untuk menghasilkan informasi yang berguna bagi pengguna program tersebut. Misalkan kita membuat sebuah program untuk mengelola data siswa di suatu sekolah maka kita harus mencatat nomor induk, nama dan umur dari siswa. Untuk mencatat data ini kita memerlukan apa yang disebut dengan variabel.

Variabel dapat didefinisikan sebagai tempat untuk menyimpan data yang memiliki suatu tipe data, variabel ini akan diwakili oleh suatu lokasi di memori komputer kita. Dengan menggunakan nama variabel ini kita akan dapat mengakses data yang tersimpan di lokasi memori tersebut.

C# adalah suatu bahasa pemrograman yang *strongly-typed* ini berarti bahwa semua object yang digunakan dalam program C# harus memiliki suatu tipe data yang spesifik dan variabel ini hanya dapat menyimpan data yang memiliki tipe yang sesuai. Misalnya jika kita mendeklarasikan variabel bertipe `int` maka variabel ini hanya dapat menyimpan data bilangan bulat dan tidak dapat menyimpan bilangan desimal. Selain itu C# juga dikatakan sebagai sebuah bahasa yang *type-safe*, kompiler C# akan menjamin bahwa data yang dimasukkan ke suatu variabel adalah tipe yang sesuai.

3.1.1 Nama Variabel

Berikut adalah aturan yang harus dipatuhi untuk memberikan suatu nama variabel dalam bahasa pemrograman C#.

- Nama variabel terdiri dari huruf, angka dan under score (`_`).
- Nama harus diawali dengan huruf. Under score juga dapat digunakan untuk mengawali nama suatu variabel tetapi ini tidak disarankan.
- C# adalah bahasa yang case sensitif, variabel dengan nama umur tidak sama dengan Umur.
- Keyword tidak bisa digunakan sebagai nama variabel, kecuali kalau keyword ini diawali dengan karakter `@`.

Berikut adalah contoh nama variabel yang benar dan salah.

Nama	Benar/Salah
<code>nomorInduk</code>	Benar
<code>nama_siswa</code>	Benar
<code>TotalPenjualan</code>	Benar
<code>2Date</code>	Salah; karena diawali dengan angka

public	Salah; karena public adalah keyword
total#pembelian	Salah; karena menggunakan karakter #
_total_memory	Benar; tetapi tidak dianjurkan
@int	Benar; keyword diawali dengan @

Banyak konvensi yang digunakan untuk memformat penamaan variabel ini. Ada yang disebut dengan notasi *Pascal*, dimana setiap kata yang digunakan sebagai nama variabel akan selalu dimulai dengan huruf besar. Notasi *Camel* memiliki kesamaan dengan dengan notasi *Pascal* hanya saja huruf pertama dalam notasi ini selalu dimulai dengan huruf kecil. Sedangkan notasi *Hungarian* mirip dengan notasi *Camel* tetapi setiap variabel akan dimulai dengan kode yang menyatakan tipe data dari variabel tersebut.

Penggunaan konvensi dalam penamaan variabel ini bisa disesuaikan dengan selera masing-masing, belakangan ini yang banyak digunakan adalah notasi *Camel*. Yang terpenting dari penamaan variabel ini adalah gunakanlah nama yang dapat memudahkan program untuk dibaca.

Notasi	Contoh
Pascal	NamaSi swa, Total Penj ual an
Camel	namaSi swa, total Penj ual an
Hungarian	strNamaSi swa, i ntTotal Penj ual an

3.1.2 Kategori Variabel

Variabel dalam C# dikelompokkan menjadi variabel static, variabel instance, variabel lokal, elemen array dan parameter.

Variabel static dibuat dengan mendeklarasikan suatu variabel dengan menggunakan keyword `static`. Variabel static adalah bagian dari tipe dan bukan merupakan bagian dari object, oleh karena itu variabel static hanya dapat diakses melalui tipenya. Jika kita mengakses data static melalui instance/object maka kompiler akan memberikan pesan kesalahan. Variabel static juga sering disebut dengan static field.

Variabel instance adalah variabel yang merupakan bagian dari instance suatu tipe. Variabel ini dideklasikan dalam suatu tipe tanpa menggunakan keyword `static`. Variabel ini juga sering disebut dengan instance field.

Variabel lokal adalah variabel yang dideklarasikan di dalam suatu blok, statemen `for`, `switch`, atau `using`. Sebelum nilai dari variabel lokal ini dapat diakses maka variabel ini perlu diberikan suatu nilai secara eksplisit. Kompiler akan gagal melakukan kompilasi jika variabel ini diakses pada saat belum diberikan suatu nilai.

```
public class FooClass
{
    private static int x;
    private int y;

    public void Method(int[] z)
    {
        string name = "Foo";
    }
}
```

```
}
```

x merupakan variabel static bertipe `int`, y adalah instance variabel bertipe `int` dan z adalah array yang memiliki elemen bertipe `int` dan name adalah sebuah variabel lokal.

3.1.2.1 Parameters

Suatu method dapat memiliki serangkaian parameter. Parameter ini akan mengirimkan informasi yang diperlukan oleh method untuk melakukan operasinya. Dalam C# parameter dibagi menjadi value parameter, output parameter dan reference parameter. Untuk membuat parameter output digunakan keyword `out` dan untuk membuat parameter reference digunakan keyword `ref`. Jika tidak ditentukan `out` atau `ref` maka parameter akan dikirimkan secara *by-value*, ini berarti salinan dari value akan dibuat pada saat kita mengakses suatu method.

```
using System;

public class Foo
{
    public static void FooMethod(int x, out int y, ref int z)
    {
        x = x * 10;
        y = x + 10;
        z = x * 100;
    }

    public static void Main()
    {
        int a = 10;
        int b;
        int c = 10;
        FooMethod(a, out b, ref c);
        Console.WriteLine("Nilai a = {0}", a);
        Console.WriteLine("Nilai b = {0}", b);
        Console.WriteLine("Nilai c = {0}", c);
    }
}
```

Pada contoh program diatas, pada method `FooMethod`, a merupakan parameter value, b merupakan parameter output dan c merupakan parameter reference.

Parameter value dan parameter reference sebelum dikirimkan kedalam suatu method harus diberikan suatu nilai, sedangkan parameter output tidak perlu diberikan suatu nilai pada saat dikirimkan kedalam suatu method, tetapi parameter output ini harus diberikan suatu nilai sebelum keluar dari *exit point* di method tersebut. Dalam contoh diatas juga terlihat bahwa untuk mengakses method yang menerima parameter `out` dan `ref` maka ekspresi untuk memanggil juga harus menggunakan keyword `out` dan `ref`.

Jika program diatas dijalankan akan menampilkan hasil berikut:

```
Nilai a = 10
Nilai b = 110
Nilai c = 10000
```

Karena a dikirimkan *by value* maka walaupun x didalam FooMethod nilainya dikalikan dengan 10 hasil yang ditampilkan tetap 10. Sedangkan c dikirimkan *by-reference* maka hasil yang ditampilkan adalah 10000; karena di dalam FooMethod nilai z diubah dengan mengalikannya dengan x yang sebelumnya telah dikalikan dengan 10 ($10 * 10 * 100$). b dikirimkan sebagai parameter output, jika b tidak diberikan suatu nilai dalam FooMethod kompilator akan memberikan pesan kesalahan berikut pada saat program di kompilasi.

“The out parameter 'b' must be assigned to before control leaves the current method”

3.1.3 Mendeklarasikan Variabel

Variabel dibuat dengan mendeklarasikan tipe data, memberinya suatu nama. Adapun cara untuk mendeklarasikan suatu variabel adalah:

type identifier;

```
int jumlahSiwa;
```

Akan mendeklarasikan variabel jumlahSiwa yang bertipe int. Kita juga dapat mendeklarasikan beberapa variabel yang memiliki tipe data yang sama dengan memisahkan setiap variabel dengan tanda koma.

```
int day, month, year;
```

Contoh diatas akan mendeklarasikan tiga variabel sekaligus yaitu day, month dan year. Pendeklarasian variabel sebaiknya dibuat secara terpisah untuk setiap variabel, ini dapat memudahkan kita untuk membaca program dan melakukan pelacakan kesalahan dalam program kita.

Pendeklarasian suatu variabel dapat juga digabungkan dengan pemberian initial value untuk variabel tersebut.

type identifier = initial_value;

```
int day = 30, month = 12, year = 1976;
```

Contoh diatas menambahkan nilai awal terhadap tiga variabel yang dideklarasikan.

Program berikut memberi contoh pendeklarasian variabel yang dilanjutkan dengan memberikan suatu nilai.

```
using System;
class Variabel
{
    public static void Main(string[] args)
    {
        int x = 10;
        Console.WriteLine("x = {0}", x);
        x = 20;
        Console.WriteLine("x = {0}", x);
    }
}
```

```
}
```

Jika program ini dijalankan maka di layar akan ditampilkan hasil sebagai berikut:

```
x = 10  
x = 20
```

Program diatas mendeklarasikan sebuah variabel bertipe `int` dengan nama `x` yang kemudian diberikan nilai 10 yang kemudian ditampilkan. Setelah itu nilai `x` diubah menjadi 20 dan kemudian ditampilkan sekali lagi.

3.1.4 Assigment / Pemberian Nilai

C# mengharuskan sebuah variabel memiliki suatu nilai sebelum dapat membaca nilai yang dimilikinya, tetapi ini tidak mengharuskan suatu variabel diinisialisasi pada saat pendeklarasiannya. Keharusan pemberian nilai kedalam variabel ini di dalam C# ini disebut dengan *definite assignment*.

Adapun cara untuk memberikan nilai kedalam suatu variabel adalah:

```
identifier = value;
```

Identifier adalah nama dari variabel sedangkan *value* adalah sebuah nilai yang diberikan kepada variabel ini.

Sebagai contoh jika program pada sub deklarasi sebelumnya dimodifikasi menjadi sebagai berikut:

```
using System;  
class Variabel  
{  
    public static void Main(string[] args)  
    {  
        int x;  
        Console.WriteLine("x = {0}", x);  
        x = 20;  
        Console.WriteLine("x = {0}", x);  
        X = 30;  
        Console.WriteLine("x = {0}", x);  
    }  
}
```

Pada contoh diatas variabel `x` coba untuk ditampilkan padahal belum memiliki suatu nilai, maka sewaktu program dikompilasi akan memberikan pesan kesalahan sebagai berikut:

```
Variabel.cs(8, 42): error CS0165: Use of unassigned local variable 'x'
```

Untuk mencegah terjadinya kesalahan diatas maka `x` harus diberi nilai sebelum ditampilkan. Pada program diatas kita dapat menambahkan 1 baris kode yaitu:

```
x = 10;
```


tepat dibawah pendeklarasian variabel x, kompile ulang program dan tidak akan muncul pesan kesalahan. Nilai dari suatu variabel dapat diubah kapan saja dengan memberikan nilai baru pada variabel tersebut.

3.1.5 Default Value

C# mengenal dua buah tipe yang disebut dengan *value-type* dan *reference-type*. Untuk tipe value nilai default-nya sesuai dengan nilai dari masing-masing tipe yang dihasilkan oleh default constructor dari tipe ini. Sedangkan untuk tipe reference nilai default-nya adalah nul l .

Variabel yang dideklarasikan sebagai variabel static, variabel instance dan elemen array akan langsung diberikan nilai default jika nilainya tidak ditentukan pada saat deklarasi. Tabel berikut memberikan gambaran yang lebih jelas mengenai nilai default dari suatu variabel

Tipe	Nilai Default
Bool	False
Char	'\0'
Enum	0
Numerik	0
Reference	null

3.1.6 Tipe Data Numerik

Tabel berikut memberikan penjelasan singkat tentang tipe data C#, padanannya dengan tipe data pada .Net Framework dan nilai yang dapat disimpannya.

Tipe Data C#	Tipe Data Net	Byte	Nilai Minimum	Nilai Maksimum
sbyte	System.S byte	1	-128	127
byte	System.B yte	1	0	255
short	System.I nt16	2	-32,768	32,767
ushort	System.U Int16	2	0	65,535
int	System.Int 32	4	-2,147,483, 648	2,147,483, 647
uint	System.UIn t32	4	0	4,294,967, 295
long	System.Int 64	8	-9,223,372, 036, 854,775,80 8	9,223,372, 036, 854,775,80 7
ulong	System.UIn t64	8	0	18,446,744 ,073 709,551,61 5
char	System.Ch ar	2	0	65,535

float	System.Single	4	1.5×10^{-45}	3.4×10^{38}
double	System.Double	8	5.0×10^{-324}	1.7×10^{308}
bool	System.Boolean	1	false (0)	true (1)
decimal	System.Decimal	16	1.0×10^{-28}	7.9×10^{28}

3.1.7 Contoh Program

Berikut adalah sebuah program yang berfungsi untuk menghitung umur dalam hari, jam dan menit. Program ini akan meminta anda untuk memasukkan data berupa tanggal, bulan dan tahun kelahiran anda. Kemudian akan melakukan perhitungan untuk mendapatkan hari, jam dan menit umur anda sampai hari ini.

```
using System;

public class Umur
{
    public static void Main()
    {
        int day;
        int month;
        int year;

        Console.Out.WriteLine("Menghitung Umur      ");
        Console.Out.WriteLine("=====");
        try
        {
            Console.Out.WriteLine("Masukkan Tanggal : ");
            day = Convert.ToInt32(Console.In.ReadLine());

            Console.Out.WriteLine("Masukkan Bulan   : ");
            month = Convert.ToInt32(Console.In.ReadLine());

            Console.Out.WriteLine("Masukkan Tahun   : ");
            year = Convert.ToInt32(Console.In.ReadLine());

            DateTime birthDate = new DateTime(year, month, day);
            DateTime today = DateTime.Now;

            TimeSpan age = today.Subtract(birthDate);
            Console.Out.WriteLine(
                "Saat ini umur anda adalah: {0} hari, {1} jam, " +
                "{2} minutes.", age.Days, age.Hours, age.Minutes);
        }
        catch (FormatException)
        {
            Console.Out.WriteLine(
                "Data tanggal, bulan dan tahun harus berupa angka.");
        }
        catch (Exception e)
        {
            Console.Out.WriteLine("Terjadi kesalahan: " + e.Message);
        }
    }
}
```

Langkah pertama dalam program ini adalah mendeklarasikan variabel `day`, `month` dan `year` yang bertipe `int`. Kemudian program akan menampilkan kalimat agar anda memasukkan data tanggal, bulan dan tahun kelahiran anda.

Kemudian dengan menggunakan `Console.ReadLine()` program dapat membaca data yang anda masukkan. Data yang dibaca dengan method `ReadLine()` ini bertipe `string`, oleh karena itu kita harus mengkonversinya menjadi tipe `int` dengan menggunakan method `ToInt32()` yang terdapat di class `System.Convert`.

Setelah mendapatkan semua informasi yang dibutuhkan kita akan membuat dua buah object bertipe `System.DateTime`, yang pertama merupakan object yang mewakili tanggal kelahiran anda dan yang kedua merupakan object yang mewakili waktu saat ini.

Kemudian kalkulasi akan dilakukan dengan mengurangi tanggal sekarang dengan tanggal kelahiran anda untuk mendapatkan umur anda. Untuk melakukan kalkulasi ini digunakan object dari class `System.TimeSpan`.

Pada langkah terakhir hasil kalkulasi akan ditampilkan dengan menggunakan `Console.WriteLine()`.

Contoh hasil dari eksekusi program diatas adalah sebagai berikut:

```
Menghitung Umur
=====
Masukkan Tanggal : 30
Masukkan Bulan   : 12
Masukkan Tahun   : 1976
Saat ini umur anda adalah: 10150 hari , 20 jam, 14 minutes.
```

3.2 Ekspresi

Ekspresi terbentuk dari rangkaian operator dan operand. Operator yang terdapat dalam suatu ekspresi menyatakan proses apa yang akan dilakukan pada suatu operand. Contoh dari operator adalah `+`, `-`, `*`, `/` dan `new`. Sedangkan contoh dari operand adalah literal, fields, variabel lokal dan ekspresi, suatu ekspresi dapat digunakan untuk membentuk ekspresi yang lain yang lebih besar.

C# memiliki tiga macam operator: operator unary, binary dan ternary. Operator unary memiliki satu buah operand dan menggunakan notasi prefix atau postfix (misalnya `--x` atau `x++`). Operator binary menggunakan dua buah operand dan menggunakan notifikasi infix (misalnya `x + y`). C# hanya memiliki satu buah operator ternary yaitu, `?:`, operator ini memerlukan tiga buah operand dan menggunakan notasi infix (misalnya `x ? y : z`).

Dalam pengoperasiannya operator memiliki tingkatan, misalnya jika kita memiliki ekspresi `x + y * z`, maka ekspresi ini akan di eksekusi dengan urutan `x + (y * z)` karena operator `*` memiliki tingkatan yang lebih tinggi dibandingkan dengan operator `+`. Jika kita ingin mengubah urutan eksekusinya, kita dapat menggunakan tanda kurung. Bahasa pemrograman C# memiliki kemampuan untuk melakukan overload terhadap operator yang berarti setiap operator bisa memiliki fungsi yang berbeda untuk object yang berbeda.

Tabel berikut menunjukkan tingkat dari operator dari level yang tinggi ke level terendah, operator dalam kategori yang sama memiliki tingkat yang sama pula.

Kategori	Ekspresi	Keterangan
Primary	<code>x.m</code>	Mengakses member
	<code>x(...)</code>	Pemanggilan method dan delegate
	<code>x[...]</code>	Mengakses array dan indexer
	<code>x++</code>	Post-increment (Penjumlahan dengan 1 setelah x dievaluasi)
	<code>x--</code>	Post-decrement (Pengurangan dengan 1 setelah x dievaluasi)
Kategori	Ekspresi	Keterangan
	<code>new T(...)</code>	Pembuatan object dan delegate
	<code>new T[...]</code>	Pembuatan array
	<code>typeof(T)</code>	Mendapatkan object System.Type dari T
	<code>checked(x)</code>	Pengecekan ekspresi dalam konteks checked
	<code>unchecked(x)</code>	Pengecekan ekspresi dalam konteks unchecked
Unary	<code>+x</code>	Identity
	<code>-x</code>	Negation
	<code>!x</code>	Logical negation
	<code>~x</code>	Bitwise negation
	<code>++x</code>	Pre-increment (Penjumlahan dengan 1 sebelum x dievaluasi)
	<code>--x</code>	Pre-decrement (Pengurangan dengan 1 sebelum x dievaluasi)
	<code>(T)x</code>	Mengkonversi x kedalam tipe T secara eksplisit
Multiplicative	<code>x * y</code>	Perkalian
	<code>x / y</code>	Pembagian
	<code>x % y</code>	Sisa pembagian
Additive	<code>x + y</code>	Penjumlahan, pengabungan string, kombinasi delegate
	<code>x - y</code>	Pengurangan, penghapusan delegate
Shift	<code>x << y</code>	Shift left
	<code>x >> y</code>	Shift right
Relational and type testing	<code>x < y</code>	Lebih kecil dari
	<code>x > y</code>	Lebih besar dari
	<code>x <= y</code>	Lebih kecil atau sama dengan
	<code>x >= y</code>	Lebih besar atau sama dengan
	<code>x is T</code>	Return true jika x adalah T, atau false jika sebaliknya
	<code>x as T</code>	Return x sebagai tipe T; atau null jika x tidak bertipe T
Equality	<code>x == y</code>	Sama
	<code>x != y</code>	Tidak sama
Logical AND	<code>x & y</code>	Integer bitwise AND, boolean logical AND
Logical XOR	<code>x ^ y</code>	Integer bitwise XOR, boolean logical XOR

Logical OR	x y	Integer bitwise OR, boolean logical OR
Conditional AND	x && y	y akan dievaluasi jika x bernilai true
Conditional OR	x y	y akan dievaluasi jika x bernilai false
Conditional	x ? y : z	y akan dievaluasi jika x bernilai true dan z akan dievaluasi jika x bernilai false
Assignment	x = y	Pemberian suatu nilai
	x op= y	Compound assignment. Bisa digunakan pada operator *= /= %= += -= <<= >>= &= ^= =

3.2.1 Checked vs Unchecked

Operator checked dan unchecked dapat digunakan dalam konteks operasi aritmatika untuk bilangan bulat. Jika kita membuat suatu program kemudian kita memberikan nilai yang lebih besar dari apa yang dapat ditampung oleh variabel tersebut maka error akan terjadi pada program kita. Ada saatnya dimana kita ingin supaya kompiler tidak memberikan error pada saat kita melakukan operasi ini, untuk itu kita dapat menggunakan operator unchecked.

Dalam konteks checked, System.OverflowException akan terjadi pada saat runtime jika kita memberikan nilai yang lebih besar kedalam suatu variabel, atau akan terjadi kesalahan pada waktu kompilasi jika operasi yang dievaluasi berupa ekspresi konstan.

```
public static void Main()
{
    int i = int.MaxValue;
    checked
    {
        Console.WriteLine(i + 1); // Exception
    }

    unchecked
    {
        Console.WriteLine(i + 1); // Overflow
    }
}
```

Jika program diatas dijalankan blok checked akan menghasilkan kesalahan karena mencoba untuk memberikan nilai sebesar nilai maksimal untuk tipe int ditambah satu.

Unhandled Exception: System.OverflowException: Arithmetic operation resulted in an overflow.

Sedangkan blok unchecked tidak akan menghasilkan kesalahan. Operasi unchecked ini mirip dengan cara kerja ordometer pada kendaraan kita, jika nilai maksimumnya sudah tercapai maka dia akan mulai lagi dari awal, misalnya jika nilai maksimumnya adalah 999,999 maka berjalan sejauh 1 kilometer lagi ordometer akan menunjukkan nilai 000,000, yang merupakan nilai minimal ordometer. Demikian juga

dengan `int.MaxValue + 1` akan menjadi nilai `int.MinValue` (-2147483648) yang merupakan nilai minimal dari tipe data `int`.

4. Flow Control

Panji Aryaputra

Aplikasi komputer tidak akan banyak berguna jika hanya bisa menjalankan satu *flow*. Bayangkan jika program yang Anda buat hanya menjalankan perintah satu persatu dari atas ke bawah. Jika programnya seperti itu, berarti cuma ada satu variasi masalah yang bisa ditangani. Berubah sedikit saja kondisinya, program tersebut sudah tidak bisa berfungsi lagi. Penanganan berbagai variasi masalah menuntut agar bahasa pemrograman memiliki *flow control*. Flow control memungkinkan program menentukan kode mana yang akan dijalankan berdasarkan kondisi-kondisi tertentu.

Sebelum melangkah lebih jauh lagi, ada istilah-istilah yang perlu dipahami sehubungan dengan flow control ini, yaitu kondisi dan aksi.

4.1 Kondisi dan Aksi

Kondisi adalah bagian dari flow control yang menentukan bagian mana yang akan dijalankan selanjutnya. Kondisi bernilai boolean (true/false). dan diapit dalam tanda kurung, seperti contoh berikut:

```
(hari == "Minggu")
```

Perhatikan:

Operator kesamaan dalam C# adalah ==, sama dengan yang digunakan dalam C. Bedanya, C# compiler akan menampilkan compile error kalo kita menggunakan =, bukan ==.

Tanda kurung merupakan keharusan. Tidak seperti VB, compiler C# akan memprotes (baca: compilation error) jika kondisi tidak diletakkan dalam tanda kurung.

Nilai kondisi harus selalu bertipe boolean. Dalam C, nilai kondisi bisa bertipe apa saja.

Aksi merupakan satu atau sekumpulan perintah yang akan dijalankan bila kondisinya terpenuhi. Bila perintah yang ingin dijalankan ada lebih dari satu, gunakan kurung kurawal untuk mengapitnya, seperti pada contoh di bawah:

```
using System;
namespace org.gotdotnet.otak
{
    class ContohKondisiAksi
    {
        public static void Main()
        {
            if (DateTime.Now.Hour > 20)
            {
```

```
        Console.WriteLine("Saatnya cuci kaki dan bobo  
!!!");  
        Console.WriteLine("Selamat malam.");  
    }  
    Console.ReadLine();  
} } }
```

4.2 Selection Statement

Selection statement digunakan untuk menentukan bagian mana dari program yang akan dieksekusi selanjutnya. C# menyediakan dua jenis selection statement, yaitu *if* dan *switch* statement.

4.2.1 if

If statement digunakan untuk mengeksekusi kode program jika kondisi tertentu terpenuhi. Misalnya dalam hal melakukan operasi pembagian, program harus mengecek terlebih dahulu apakah pembagi bernilai 0 atau tidak agar tidak terjadi runtime error.

```
using System;  
  
namespace org.gotdotnet.otak  
{  
    class ContohIf  
    {  
        public static void Main2()  
        {  
            Console.WriteLine("Masukkan nilai x : ");  
            int x = int.Parse(Console.ReadLine());  
  
            Console.WriteLine("Masukkan nilai y : ");  
            int y = int.Parse(Console.ReadLine());  
  
            if (y!=0)  
            {  
                double hasil = x/y;  
                Console.WriteLine("Hasil pembagian x/y =  
{0}",  
                    hasil);  
            }  
  
            Console.ReadLine();  
        }  
    }  
}
```

Dalam contoh di atas, operasi pembagian dilakukan hanya jika y tidak bernilai 0. Bagaimana jika y bernilai 0? Program tidak akan mengeksekusi apa-apa jika nilai y sama dengan 0.

4.2.2 if-else

Satu variasi dari if statement adalah if-else. Dalam statement ini, selain menentukan langkah apa yang harus dilakukan jika suatu kondisi terpenuhi, kita juga bisa menentukan langkah apa yang mesti dilakukan kalau kondisi tersebut TIDAK terpenuhi. Masih berhubungan dengan contoh sebelumnya, jika y bernilai 0, kita ingin mencetak pesan ke layar untuk memberitahukan kepada user bahwa operasi pembagian tidak bisa dilakukan karena pembagi bernilai 0.

```
using System;

namespace org.gotdotnet.otak
{
    class ContohIfElse
    {
        public static void Main()
        {
            Console.WriteLine("Masukkan nilai x : ");
            int x = int.Parse(Console.ReadLine());

            Console.WriteLine("Masukkan nilai y : ");
            int y = int.Parse(Console.ReadLine());

            if (y!=0)
            {
                double hasil = x/y;
                Console.WriteLine("Hasil pembagian x/y = {0}", hasil);
            }
            else
            {
                Console.WriteLine("Error: y bernilai 0");
            }

            Console.ReadLine();
        }
    }
}
```

4.2.3 switch

Seringkali dalam program kita ingin melakukan pengecekan untuk beberapa kemungkinan nilai dari suatu variable. Bila ini dilakukan dengan menggunakan if, bentuknya akan seperti berikut:

```
using System;

namespace org.gotdotnet.otak
{
    class ContohIfElseIf
    {
        public static void Main()
        {
            if (DateTime.Now.DayOfWeek==DayOfWeek.Sunday)
            {
                Console.WriteLine("Sekarang hari Minggu");
            }
            else if (DateTime.Now.DayOfWeek==DayOfWeek.Monday)
            {
                Console.WriteLine("Sekarang hari Senin");
            }
        }
    }
}
```

```
    }
    else if (DateTime.Now.DayOfWeek==DayOfWeek.Tuesday)
    {
        Console.WriteLine("Sekarang hari Selasa");
    }
    else
    {
        Console.WriteLine("Sekarang hari apa ya?");
    }
    Console.ReadLine();
}
}
```

Walaupun cara di atas dapat digunakan dan benar, ada cara lain yang lebih sederhana yaitu dengan menggunakan switch. Contohnya adalah sbb.:

```
using System;

namespace org.gotdotnet.otak
{
    class ContohSwitch
    {
        public static void Main()
        {
            switch (DateTime.Now.DayOfWeek)
            {
                case DayOfWeek.Sunday:
                {
                    Console.WriteLine("Sekarang hari Minggu");
                    break;
                }

                case DayOfWeek.Monday:
                {
                    Console.WriteLine("Sekarang hari Senin");
                    break;
                }
                case DayOfWeek.Tuesday:
                {
                    Console.WriteLine("Sekarang hari Selasa");
                    break;
                }
                default:
                {
                    Console.WriteLine("Sekarang hari apa ya?");
                    break;
                }
            }

            Console.ReadLine();
        }
    }
}
```

Perhatikan hal-hal berikut dari contoh di atas:
Variabel yang dicek bisa bernilai angka atau string. Bandingkan dengan C yang hanya bisa menggunakan variabel bertipe angka.

Keyword *break* harus digunakan untuk setiap case. Dalam C, *break* tidak mesti digunakan.

Bagian default dari *switch* berfungsi seperti *else* dalam *if*, artinya jika tidak ada kecocokan dalam case-case lainnya, maka bagian dalam case default yang akan dieksekusi. Bagian ini sebaiknya dibiasakan untuk diisi.

4.3 Iteration Statement

Salah satu statement terpenting yang harus dimiliki suatu bahasa pemrograman adalah *iteration statement*. Statement jenis ini digunakan untuk menentukan bagian mana dari program yang akan dieksekusi berulang-ulang dan apa kondisi yang menentukan perulangan tersebut. Dalam C# ada empat iteration statement yang dapat digunakan, yaitu: *while*, *do*, *for*, dan *foreach*.

4.3.1 while

While statement berguna untuk melakukan perulangan selama kondisi bernilai true. Karena pengecekan kondisinya dilakukan di awal maka ada kemungkinan badan loop tidak akan dijalankan sama sekali.

```
using System;
namespace org.gotdotnet.otak
{
    class ContohWhile
    {
        public static void Main()
        {
            int i = 0;
            while (i < 10)
            {
                if (i % 2 == 0)
                {
                    Console.WriteLine("Angka genap: " + i);
                }
                i += 1;
            }
            Console.ReadLine();
        }
    }
}
```

Dalam contoh di atas, selama *i* masih bernilai lebih kecil dari 10, badan loop akan dijalankan berulang-ulang. Badan loop sendiri isinya adalah mencetak bilangan-bilangan genap.

4.3.2 do

Do memiliki fungsi yang mirip dengan while, yaitu untuk melakukan perulangan. Contoh:

```
using System;
namespace org.gotdotnet.otak
{
    class ContohDo
    {
        public static void Main()
        {
            int i = 0;
            do
            {
                if (i%2==0)
                {
                    Console.WriteLine("Angka genap: " + i);
                }
                i += 1;
            }
            while (i < 10);
            Console.ReadLine();
        }
    }
}
```

Perhatikan bahwa berbeda dengan *while*, pengecekan kondisi dalam *do* dilakukan di akhir. Ini berarti bahwa badan loop akan dijalankan minimal sekali.

4.3.3 for

Salah satu bentuk perulangan dalam C# adalah for. Fungsi konstruksi ini sama dengan for dalam bahasa-bahasa lain yang diturunkan dari C. For digunakan untuk melakukan perulangan yang didasarkan atas nilai diskrit, misalnya integer. Salah satu penggunaan for yang paling umum adalah dalam menelusuri suatu array, seperti dalam contoh di bawah.

```
using System;
namespace org.gotdotnet.otak
{
    class ContohFor
    {
        public static void Main()
        {
            string[] drives = System.Environment.GetLogicalDrives();
            for (int i=0; i < drives.Length; i++)
            {
                Console.WriteLine("drive " + drives[i]);
            }
            Console.ReadLine();
        }
    }
}
```

Pada contoh di atas, array drives berisi daftar logical drive yang ada dan kemudian for statement digunakan untuk mencetak nama drive tersebut satu persatu.

4.3.4 foreach

Satu bentuk iterasi khusus yang tidak berasal dari C adalah foreach. Bentuk ini sebenarnya diambil dari Visual Basic (*for each*). Statement foreach digunakan untuk menelusuri suatu collection, misalnya array. Contoh:

```
using System;
namespace org.gotdotnet.otak
{
    class ContohForeach
    {
        public static void Main()
        {
            string[] drives = System.Environment.GetLogicalDrives();
            foreach (string drive in drives)
            {
                Console.WriteLine("drive " + drive);
            }
            Console.ReadLine();
        }
    }
}
```

Kode 1: Contoh Penggunaan foreach

Perhatikan dalam contoh di atas bahwa variabel *drive* secara implisit bertipe read only. Jadi bila ada statement yang berusaha mengubah nilai variabel hari, maka compiler C# akan menampilkan pesan kesalahan sewaktu melakukan kompilasi.

4.4 Jump Statement

Jump statement digunakan untuk mentransfer kontrol eksekusi dari suatu bagian ke bagian lain dalam program. Beberapa statement yang termasuk dalam jenis ini sebaiknya diminimalkan penggunaannya, contohnya goto, continue, dan break. Alasannya adalah karena penggunaan statement-statement tersebut mengurangi kejelasan flow program. Selain itu pula, kemungkinan terjadinya bug juga semakin besar terutama jika bagian-bagian yang akan dilompati tersebut memiliki bagian inisiasi dan terminasi.

4.4.1 break

Statement ini digunakan untuk “melompat” keluar dari while, for, dan switch statement yang sudah dibahas sebelumnya. Berikut ini adalah satu contoh penggunaan break:

```
using System;
namespace org.gotdotnet.otak
{
    class ContohBreak
    {
        public static void Main()
        {
            for (int i=0; i<10; i++)
            {
                Console.WriteLine("i=" + i);
                if (i==5) break;
            }
            Console.ReadLine();
        }
    }
}
```

Dalam contoh di atas, walaupun bentuk perulangan for di atas dispesifikasikan untuk dijalankan sebanyak 10 kali, pada saat nilai mencapai 5, break dijalankan. Akibatnya, eksekusi akan “melompat” keluar dari badan perulangan. Jadi pada kenyataannya, perulangan hanya dilakukan sebanyak 6 kali.

Tips: gunakan model for semacam ini untuk melakukan perulangan yang jumlah maksimum iterasinya sudah diketahui di awal dan ada kondisi-kondisi tertentu yang mungkin menyebabkan iterasi berhenti sebelum mencapai jumlah maksimumnya. Dengan cara ini, iterasi pasti akan berhenti setelah dijalankan n-kali. Bila yang digunakan adalah bentuk *while (kondisi)* kemungkinan terjadinya *infinite loop* lebih besar.

4.4.2 continue

Continue dapat digunakan dalam semua struktur perulangan, misalnya for dan while. Statement continue ini berfungsi untuk melanjutkan eksekusi program ke iterasi berikutnya dengan “melompati” statement-statement berikutnya dalam badan loop. Perhatikan contoh berikut:

```
using System;
namespace org.gotdotnet.otak
{
    class ContohContinue
    {
        public static void Main()
        {
            for (int i=0; i<10; i++)
            {
                if (i==5) continue;
                Console.WriteLine("i=" + i);
            }
        }
    }
}
```

```
        Console.ReadLine();
    }
}
```

Pada contoh di atas, bila `i==5`, perintah `continue` akan dijalankan. Akibatnya statement berikutnya dalam badan loop (`Console.WriteLine`) akan diabaikan dan eksekusi program akan dilanjutkan pada iterasi berikutnya (`i=6`).

4.4.3 goto

Goto digunakan untuk melanjutkan eksekusi program di label yang sudah didefinisikan sebelumnya. Jika jump statement lainnya penggunaannya terbatas, goto ini termasuk yang lebih “bebas”, dalam arti kata ia bisa digunakan untuk melakukan lompatan ke bagian mana saja dari program kecuali melompat ke dalam suatu blok.

Berikut adalahh contoh penggunaan goto:

```
using System;

namespace org.gotdotnet.otak
{
    class ContohGoto
    {
        public static void Main()
        {
            for (int i=0; i<10; i++)
            {
                if (i==5) goto selesai;
                Console.WriteLine("i=" + i);
            }
            selesai:
            Console.ReadLine();
        }
    }
}
```

Statement ini termasuk yang agak kontroversial. Hampir semua buku kuliah dan dosen akan mengharamkan penggunaan goto dalam program. Tidak sedikit orang yang “protes” terhadap penyertaan statement ini dalam C#. Well, itu di luar wilayah bahasan buku ini. :)

4.4.4 return

Seperti halnya dalam bahasa turunan C lainnya, perintah `return` digunakan untuk mengembalikan kontrol eksekusi ke pemanggil. Misalnya method A memanggil method B, pada saat perintah `return` dijalankan dalam method B, maka kontrol eksekusi akan dikembalikan ke method A dan perintah selanjutnya yang akan dijalankan adalah perintah berikutnya dalam method A. Contoh:

```

using System;
namespace org.gotdotnet.otak
{
    class ContohReturn
    {
        public static void FungsiB()
        {
            Console.WriteLine("Fungsi B");
            Console.WriteLine("Akan menjal ankan return ...");
            return;
        }

        public static void FungsiA()
        {
            Console.WriteLine("Fungsi A");
            Console.WriteLine("Akan memanggil l Fungsi B");
            FungsiB();
            Console.WriteLine("Mel anjutkan      perintah      Fungsi A
berikutnya");
        }

        public static void Main()
        {
            FungsiA();
            Console.ReadLine();
        }
    }
}

```

Ada dua jenis penggunaan return, yaitu return dengan menyertakan suatu nilai dan return tanpa nilai. Return tanpa nilai digunakan dalam method yang tipe nilai kembaliannya void, dengan kata lain method tersebut tidak mengembalikan nilai. Dalam hal ini return hanya berfungsi untuk mengembalikan kontrol eksekusi ke pemanggil.

Jenis penggunaan yang kedua, return dengan menyertakan nilai, selain mengembalikan kontrol eksekusi, juga akan mengembalikan nilai tersebut ke pemanggil. Tipe nilai yang dikembalikan oleh return harus sama dengan tipe nilai kembalian dari method.

4.4.5 throw

Penggunaan *throw* ini berkaitan dengan penanganan error (try statement) dalam C#. Throw digunakan untuk membangkitkan *exception* dalam program. Untuk lebih jelasnya perhatikan contoh berikut.

```

using System;
namespace com.gotdotnet.otak
{
    class ContohThrow
    {
        public static void Main()
        {
            try
            {
                Console.Wri te("Ketik nama Anda: ");
                string nama = Console.e.ReadLine();
                i f (nama!="ri sman")
                    throw new System.Appl icati onExcepti on("Nama ti dak
                    di kenal");
            }
        }
    }
}

```



```
        Console.WriteLine("Selamat, Anda termasuk orang terkenal");
    });
    }
}
catch(ApplicationException ae)
{
    Console.WriteLine("Exception: " + ae.Message);
}
Console.ReadLine();
}
```

Dalam program di atas proses utamanya diletakkan dalam blok *try*, kemudian disertai oleh blok *catch*. Ini berarti jika ada exception yang ditimbulkan dalam blok *try*, eksekusi program akan melompat ke block *catch* (jika tipe exceptionnya sesuai).

Selain dihasilkan/dibangkitkan oleh sistem, misalnya *DivideByZeroException* dan *IndexOutOfRangeException*, exception juga bisa dibangkitkan oleh aplikasi. Pada contoh di atas, kondisi *nama!="risman"* itu sendiri pada dasarnya bukan merupakan exception namun aplikasi ingin memperlakukan itu sebagai exception. Untuk keperluan inilah perintah *throw* digunakan. Jadi jika *nama!="risman"*, maka aplikasi akan membangkitkan exception sehingga eksekusi program akan melompat ke bagian penanganan exception (blok *catch*).

Exception dijelaskan secara lebih detail pada Bab VI: Debugging dan Penanganan Error.

5. methods

Adi Wirasta

5.1 Pengenalan Method

Method adalah bagian dari tubuh program yang mengimplementasikan suatu action sehingga class atau object dapat bekerja. Method diimplementasikan didalam class dan menyediakan informasi tambahan yang mana class tidak dapat menangani sendiri. Sebelum dilanjutkan tentang method, mungkin perlu diingat kembali bahwa class sendiri memiliki anggota-anggota yaitu *constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors*, dan *nested type declarations*.

Method dapat didefinisikan lagi yaitu

- method yang memiliki beberapa parameter(tapi bisa kosong)
- method dapat mengembalikan nilai (kecuali tipe pengembaliannya void)
- method dapat terdiri dari static atau non-static.
 - Static : method static hanya dapat di akses dari class.
 - Non-static : dapat di akses dari instances

Contoh dari method adalah seperti ini :

```
public class Stack
{
    public static Stack Clone(Stack s) {...}
    public static Stack Flip(Stack s) {...}
    public object Pop() {...}
    public void Push(object o) {...}
    public override string ToString() {...}
    ...
}
class Test
{
    static void Main() {
        Stack s = new Stack();
        for (int i = 1; i < 10; i++)
            s.Push(i);
        Stack flipped = Stack.Flip(s);
        Stack cloned = Stack.Clone(s);
        Console.WriteLine("Original stack: " + s.ToString());
        Console.WriteLine("Flipped stack: " + flipped.ToString());
        Console.WriteLine("Cloned stack: " + cloned.ToString());
    }
}
```

Contoh diatas memperlihatkan class stack yang memiliki beberapa static methods(Clone dan Flip) dan beberapa non-static methods (Push, Pop dan ToString).

5.2 Implementasi methods

5.2.1 static methods

Method dapat di overload, artinya satu nama method bisa dipakai berkali-kali selama dia memiliki sesuatu yang unik. Sesuatu yang unik pada method tersebut dapat terdiri dari banyaknya parameter, tipe parameter atau modifier parameter yang berbeda. Contohnya seperti dibawah ini :

```
class Test
{
    static void F() {
        Console.WriteLine("F()");
    }
    static void F(object o) {
        Console.WriteLine("F(object)");
    }
    static void F(int value) {
        Console.WriteLine("F(int)");
    }
    static void F(ref int value) {
        Console.WriteLine("F(ref int)");
    }
    static void F(int a, int b) {
        Console.WriteLine("F(int, int)");
    }
    static void F(int[] values) {
        Console.WriteLine("F(int[])");
    }
    static void Main() {
        F();
        F(1);
        int i = 10;
        F(ref i);
        F(out i);
        F((object)1);
        F(1, 2);
        F(new int[] {1, 2, 3});
    }
}
```

Class diatas memiliki nama method `F` tapi memiliki jumlah, tipe dan modifier yang berbeda. Output dari class diatas jika di eksekusi adalah :

```
F()
F(int)
F(ref int)
F(object)
F(int, int)
F(int[])
```

C# dapat mengembalikan banyak nilai dari satu method dengan menggunakan out parameter.

Contohnya seperti ini :

```
Using System;
```

```
Public class OutTest
{
    Public static int Output (out int a)
    {
        a = 25;
        return 0;
    }

    public static void main ( )
    {
        int a;
        Console.Wri teLi ne (Output (out a));
        Console.Wri teLi ne (a);
    }
}
```

hasilnya adalah :

```
0
25
```

static method juga dapat mengembalikan lebih dari satu variabel dengan cara menggunakan keyword params.

Contohnya adalah seperti ini :

```
usi ng System;
publ i c class Params
(
    publ ic static void Parameter(params int[] list)
    {
        for (int x= 0; x< list.Length ; x++)
            Console.Wri teLi ne(list[x]);
        Console.Wri teLi ne();
    }

    publ ic static void Mai n()
    {
        Parameter(10, 15, 20);
    }
}
```

5.2.2 non-static method

Telah dijelaskan sebelumnya bahwa static method dapat diakses hanya melalui class sedangkan non-static method dapat diakses melalui instance.

Kita akan mencoba melihat bagaimana implementasi non-static method dibandingkan dengan static method.

Buat satu console application project baru.

ketikan di dalam class1 seperti ini (copy – paste saja semua code dibawah ini kedalam class1.cs) :

```
usi ng System;
namespace nonstatic
{
```

```
class First
{
    public virtual void one()
    {
        Console.WriteLine("first one");
    }
}
class Second: First
{
    public override void one()
    {
        Console.WriteLine("Second one");
    }
}
/// <summary>
/// Summary description for Class1.
/// </summary>
class Output
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        //
        // TODO: Add code to start application here
        //
        Second y = new Second();
        First x = y;
        x.one();
        y.one();
    }
}
}
```

output code diatas menjadi :

```
second one
second one
```

Output ini terjadi karena objek x sudah di-instance oleh objek y dimana objek y adalah class second yang turunan dari class First. (Penjelasan override dan virtual dapat dibaca pada materi Object Oriented).

jika class First dan class Second kita ubah seperti ini :

```
class First
{
    public void one()
    {
        Console.WriteLine("first one");
    }
}

class Second: First
{
    public void one2()
    {
```

```
        Console.WriteLine("Second one");  
    }  
}
```

Maka output code yang dihasilkan adalah :

```
First one  
First one
```

Output ini dihasilkan karena perubahan yang dilakukan menyebabkan class Second menjadi memiliki 2 non-static method yaitu method one dan method one2.

Lalu, jika class First dan class Second kita ubah lagi seperti ini :

```
class First  
{  
    public static void one()  
    {  
        Console.WriteLine("fi rst one");  
    }  
}  
  
class Second: First  
{  
    public void onen()  
    {  
        Console.WriteLine("Second one");  
    }  
}
```

Maka akan muncul eror seperti ini :

```
Static member 'nonstatic.First.one()' cannot be accessed with an instance reference;  
qualify it with a type name instead
```

```
Static member 'nonstatic.First.one()' cannot be accessed with an instance reference;  
qualify it with a type name instead
```

Eror ini memperlihatkan bahwa static method tidak dapat diakses menggunakan instance.

5.3. Modifier

Berapa contoh implementasi method diatas memakai modifier. Apa itu Modifier? Modifier adalah keyword yang digunakan untuk menspesifikasi deklarasi pengaksesan suatu member atau tipe.

Ada 4 modifier pengaksesan yang akan diperlihatkan :

- public
- protected
- internal
- private

Keempat level diatas akan dijelaskan pada tabel dibawah ini :

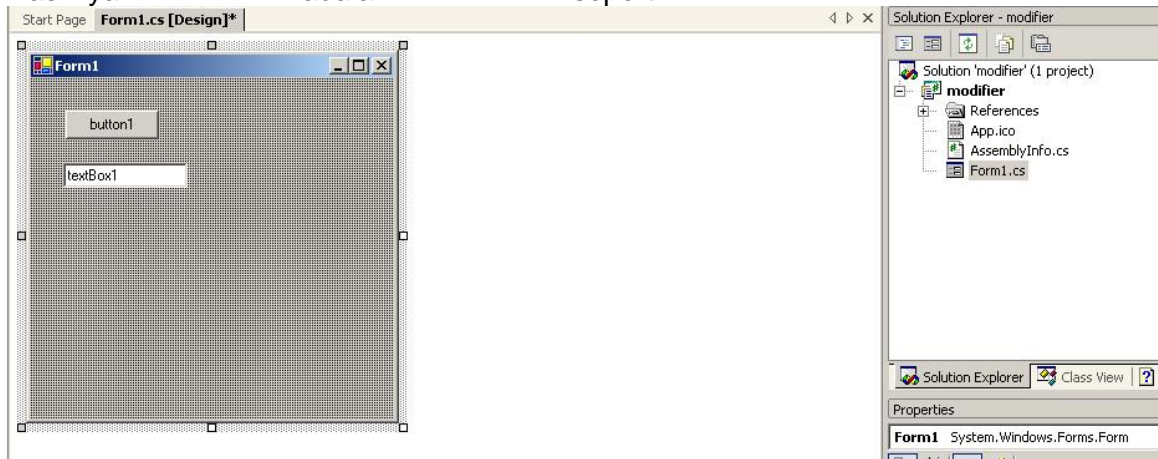
Level akses	Arti
Public	Akses tidak dibatasi
protected	Akses dibatasi pada classnya saja atau tipe yang diturunkan dari class.
internal	Akses hanya dibatasi pada satu project yang sama.
private	Akses dibatasi pada tipe.

Dengan ke empat level akses ini, method dapat diberikan pembatasan-pembatasan pengaksesan. Jika methods yang kita buat hanya ingin diakses pada project tertentu, kita bisa memberikan modifier internal.

Kita akan coba mengetahui bagaimana cara kerja dari 4 modifier diatas. Cara pemakaian yang akan kita lakukan dengan membuat sebuah projek windows application.

Langkah-langkahnya adalah sebagai berikut :

1. Buat sebuah projek baru.
2. Pilih windows application.
3. Berikan nama projeknya "modifier".
4. Sebuah projek dan satu buah windows form telah dibuat. Kita klik dan drop sebuah kontrol textbox dan sebuah kontrol button.
5. Hasilnya adalah seperti ini :



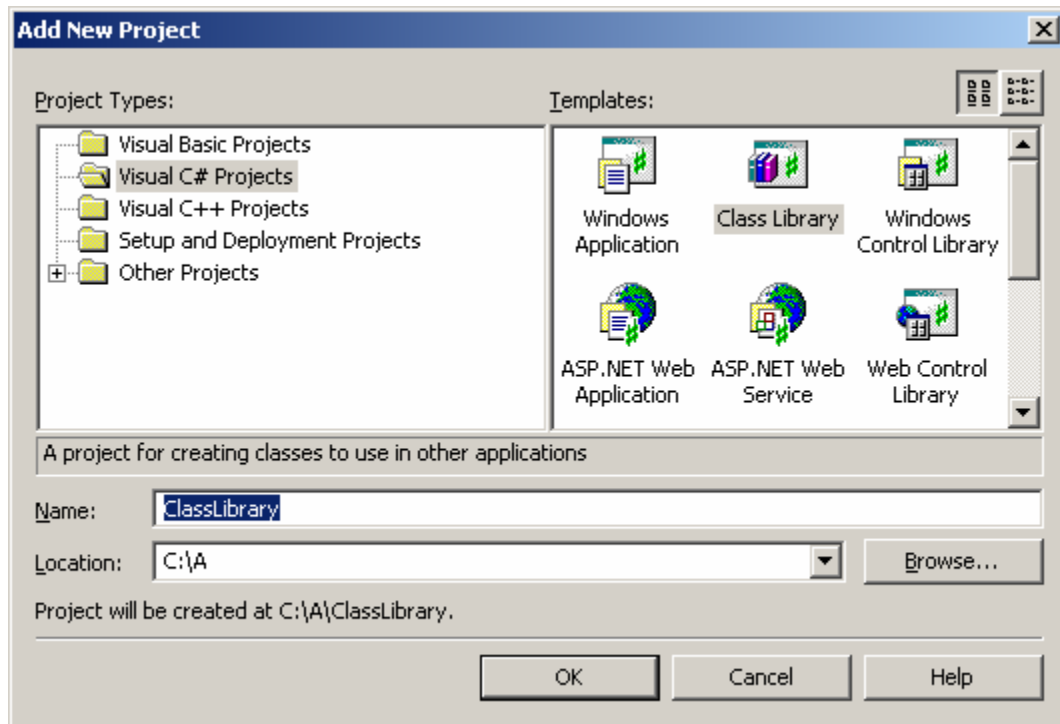
Gambar 5-1. Windows Forms

6. Klik dua kali kontrol button dan kita akan mendapati cursor telah berada di :

```
private void button1_Click(object sender, System.EventArgs e)
{
}

```

7. Selanjutnya, kita coba membuat satu projek lagi. Kita akan membuat sebuah Projek class library. Beri nama ClassLibrary



Gambar 5-2. Membuat project Class Library

8. Setelah terbentuk, kita akan menambahkan projek ClassLibrary kedalam projek modifier dengan cara Project > add reference > pilih tab project > pilih ClassLibrary pada Project Name dengan mengklik tombol Select >OK.
9. Didalam ClassLibrary, kita mendapatkan sebuah Class bernama Class1. Namespace ClassLibrary kita ganti menjadi modifier.ClassLibrary sehingga isinya adalah sebagai berikut :

```
using System;
namespace modifier.ClassLibrary
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    public class Class1
    {
        public Class1()
        {
            //
            // TODO: Add constructor logic here
            //
        }
    }
}
```

10. Tambahkan sebuah method pada Class public class Class1() dibawah method public Class1 seperti ini :

```
public string A()
{
    return "Ronald";
}
```


Kembali ke method private void button1_Click, kita isikan dengan code :

```
ClassLibrary.Class1 x = new ClassLibrary.Class1();  
this.textBox1.Text = x.A();
```

11. Klik run. Kita akan mendapatkan textbox berisi "Ronaldo".
Bagaimana jika modifier string A() pada Class Class1 diganti dengan private, internal dan protected ? ternyata masing-masing mengucapkan eror yang sama : 'modifier.ClassLibrary.Class1.A()' is inaccessible due to its protection level.
Sekarang kita akan mencoba penggunaan modifier internal. Kita akan menambahkan sebuah method dibawah method public string A () dengan code seperti ini :

```
internal string B()  
{  
    return "Morientes";  
}
```

kembali ke method private void button1_Click, kita isikan dengan code :

```
ClassLibrary.Class1 x = new ClassLibrary.Class1();  
this.textBox1.Text = x.B();
```

Ternyata eror yang keluar adalah seperti berikut : 'modifier.ClassLibrary.Class1.A()' is inaccessible due to its protection level

Hal ini terjadi karena internal hanya dapat diakses hanya didalam project ClassLibrary tersebut seperti yang telah disebutkan diatas.

Jika public string A() kita ubah menjadi seperti ini :

```
public string A()  
{  
    return B();  
}
```

Lalu method private void button1_Click, kita isikan dengan code :

```
ClassLibrary.Class1 x = new ClassLibrary.Class1();  
this.textBox1.Text = x.A();
```

Kemudian jika kita klik run lalu klik button yang memunculkan "morientes" pada textbox, maka kita dapat melihat penggunaan internal hanya dapat dipakai didalam projek tersebut.

Kita akan mencoba penggunaan protected. Dikatakan sebelumnya bahwa Akses modifier protected dibatasi pada classnya saja atau tipe yang diturunkan dari class. Kita ubah isi dari class1.cs menjadi seperti ini :

```
using System;  
namespace modifier.ClassLibrary  
{  
    public class Class1  
    {  
        public Class1()  
        {  
        }  
        protected string A()  
        {  
        }  
    }  
}
```

```
        return "Zidane";
    }
}

public class Class2 : Class1
{
    public string B()
    {
        string x;
        Class1 a = new Class1();
        x = a.A();
        return x;
    }
}
```

Lalu method private void button1_Click, kita isikan dengan code :

```
ClassLibrary.Class2 x = new ClassLibrary.Class2();
this.textBox1.Text = x.B();
```

Eror yang akan kita dapatkan adalah :

Cannot access protected member 'modifier.ClassLibrary.Class1.A()' via a qualifier of type 'Class1'; the qualifier must be of type 'Class2' (or derived from it).

Statement `x = a.A()` memunculkan eror karena `Class1` tidak diturunkan dari `Class2`. (seharusnya `Class2` diturunkan dari `Class1`).

Tapi jika isi method `public string B()` kita ubah menjadi :

```
public string B()
{
    string x;
    //Class1 a = new Class1();
    Class2 b = new Class2();
    //x = a.A();
    x = b.A();
    return x;
}
```

Maka kita akan mendapatkan isi dari textbox adalah "Zidane".

6. Debugging dan Penanganan Error

Panji Aryaputra

Program komputer yang handal harus mampu menangani dengan baik berbagai macam error yang muncul selama program dijalankan. Umumnya penanganan yang baik meliputi pencatatan error tersebut dalam *error log*, menginformasikan kepada user apa yang terjadi dan langkah apa yang bisa/harus diambil selanjutnya, dan lebih baik lagi bila program tersebut bisa *recover* sehingga user bisa melanjutkan pekerjaannya. Dalam hal ini tool yang digunakan berperan besar dalam membantu mempercepat dan mempermudah debugging (pelacakan kesalahan). Seringkali faktor pembeda antara jika menggunakan tool murah/gratis dan yang mahal terletak pada seberapa bagus fasilitas debuggingnya.

6.1 Penanganan Error

Penanganan error biasa dilakukan dengan melakukan pengecekan nilai kembalian fungsi. Cara ini populer pada bahasa-bahasa yang lebih dulu muncul, misalnya pada bahasa C. Contoh penggunaannya adalah seperti di bawah (dalam *pseudo-code*)

```
sukses = fungsi 1();  
if not(sukses) exit  
  
sukses = fungsi 2;  
if not(sukses) exit;  
  
sukses = fungsi 3();  
if not(sukses) exit  
  
dst. . . .
```

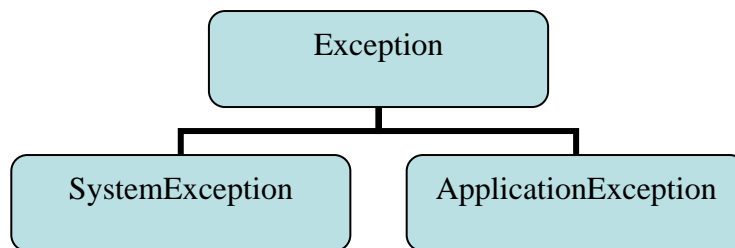
Tidak jarang kita temui lebih dari 50% kode digunakan untuk melakukan penanganan error. Anda yang terbiasa menggunakan bahasa tingkat rendah seperti C pasti sangat familiar dengan ini. Logika program yang pendek, pada saat diimplementasikan, bisa menjadi panjang karena penanganan error dengan cara ini. Tapi kekhawatiran utama bukan pada panjang kodenya, melainkan kemudahan dalam mengartikan suatu kode yang isinya berbau antara proses utama dan penanganan error. Bayangkan jika dari 200 baris kode, hanya 50 diantaranya yang memuat inti logika program. Berapa cepat kita bisa mencari dan mempelajari (apalagi mengubah) inti proses dalam kode seperti itu?

Untunglah dalam bahasa-bahasa pemrograman modern, seperti C++, Java, dan C#, telah disediakan cara baru untuk menangani error. Cara ini, yang dikenal sebagai *exception handling*, berkaitan erat dengan paradigma pemrograman berbasis object (OOP). Ide utama dari exception handling ini adalah memisahkan antara kode-kode yang berisikan inti proses dan kode-kode yang digunakan untuk menangani error.

6.2 Exception

Secara teknis *exception* adalah obyek yang merepresentasikan error yang muncul pada saat aplikasi dijalankan. Jadi frase “membangkitkan exception” berarti membuat obyek Exception baru, menginisiasi informasi-informasi di dalamnya, dan menginformasikan pada sistem bahwa ada error yang muncul.

Exception diperkenalkan dalam dunia OOP. Berbeda dengan fungsi-fungsi C yang menggunakan nomor/kode untuk membedakan jenis error, exception menggunakan **Class** yang berbeda untuk tiap jenis kesalahan. Pada level yang paling atas, Exception memiliki dua turunan, yaitu SystemException dan ApplicationException (lihat gambar di bawah). SystemException dihasilkan oleh CLR dan .Net framework, sedangkan ApplicationException digunakan untuk exception yang didefinisikan oleh aplikasi.



Gambar 6-1. Exception Hierarchy

Dari atas ke bawah, Class-Class tersebut merepresentasikan exception yang makin spesifik. Tergantung pada keperluan, aplikasi bisa memilih untuk menangani exception pada level yang lebih spesifik atau general.

6.3 Try statement

Konstruksi yang digunakan C# untuk menangani exception mirip (kalau tidak sama) dengan yang digunakan dalam C++ dan Java, yaitu *try statement*. Mari kita perhatikan struktur di bawah:

```
try
{
    // operasi-operasi yang mungkin menghasilkan
    exception
}
catch(Exception ex)
{
    // penanganan exception
}
finally
{
    // langkah-langkah yang harus selalu dilakukan,
    // ada atau tidak ada exception
}
```

Try statement terdiri dari tiga blok, yaitu blok *try*, *catch*, dan *finally*. Blok *try* memuat inti proses program (yang mungkin menghasilkan *exception*), sedangkan blok *catch* adalah blok yang akan menangani *exception-exception* yang dihasilkan. Blok *finally* sendiri isinya adalah langkah-langkah yang harus selalu dijalankan, baik bila *exception* dihasilkan atau tidak. Blok *catch* dan *finally* tidak harus ada, tetapi minimal salah satu harus menyertai blok *try*. Jadi kemungkinannya ada tiga, yaitu *try-catch*, *try-finally*, dan *try-catch-finally*.

Urutan eksekusi kode program dalam *try* statement adalah sbb.:

Statement-statement dalam blok *try* akan dieksekusi satu persatu (dari atas ke bawah) seperti biasa.

Bila tidak ada *exception* yang dihasilkan, setelah semua statement dalam blok *try* dijalankan, statement-statement dalam blok *finally* akan dijalankan.

Bila terjadi kesalahan yang menghasilkan *exception*, eksekusi program akan melompat dari baris di mana *exception* itu dihasilkan ke blok *catch* yang memiliki tipe *exception* yang sesuai. Sesuai dalam hal ini bisa berarti bahwa tipe *exception*nya (Classnya) sama atau blok *catch* tersebut menspesifikasikan *superclass* dari *exception* yang dihasilkan.

Setelah blok *catch* selesai dijalankan, eksekusi dilanjutkan ke blok *finally*.

Jadi statement-statement dalam blok *try* menjadi relatif sederhana karena tidak perlu lagi tiap kali mencek apakah statement sebelumnya menghasilkan error atau tidak. Tiap kali terjadi error, eksekusi akan melompat keluar dari blok *try*.

Berikut adalah contoh penggunaan *try* statement:

```
using System;
using System.IO;

namespace com.gotdotnet.otak
{
    class ContohTry
    {
        public static void Main()
        {
            StreamReader sr = null;

            try
            {
                Console.WriteLine("Ketik nama file: ");
                string namaFile = Console.ReadLine();

                sr = File.OpenText(namaFile);
                string s = sr.ReadToEnd();
                Console.WriteLine("**** Isi file {0}: ****",
namaFile);
                Console.WriteLine(s);
            }
            catch(Exception e)
            {
                Console.WriteLine("Exception: " + e.Message);
            }
            finally
            {
                if (sr != null)
                    sr.Close();
            }
        }
    }
}
```

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

Program di atas meminta user untuk memasukkan nama file dan kemudian program akan berusaha membaca dan mencetak isi file tersebut ke layar. Satu-satunya blok catch pada contoh di atas memiliki parameter bertipe Exception. Ini berarti semua jenis kesalahan akan ditangkap dan ditangani oleh blok tersebut. Secara umum ini bukan ide yang bagus. Blok catch sebaiknya hanya menangani exception yang spesifik, bukan semua jenis exception.

Blok finally di atas sendiri berisi statement untuk menutup StreamReader yang digunakan untuk membaca file. Ini akan banyak menyederhanakan program kalau exception mungkin dihasilkan di banyak tempat setelah StreamReader dibuka. Dengan menempatkan *sr.close* pada blok finally, ini menjamin bahwa pada berakhirnya eksekusi fungsi tersebut, StreamReader **pasti** ditutup. Hati-hati menggunakan blok finally. Pada contoh di atas, *st* dicek lebih dahulu apakah nilainya null atau tidak. Bila exception terjadi sebelum *st* dibuka, maka tidak ada yang perlu ditutup.

Satu hal lagi, exception dalam C# boleh diabaikan, tidak seperti Java yang mengharuskan si pemanggil untuk menangani semua kemungkinan exception yang dihasilkan oleh method yang dipanggil.

6.4 Multiple Catch Block

Dalam try statement, blok catch bisa ada lebih dari satu karena ada berbagai jenis exception yang harus ditangani. Jadi satu blok menangani satu jenis kesalahan (sesuai dengan tipe parameter yang dispesifikasikan).

Bila ada lebih dari satu blok catch, blok-blok tersebut harus diurut berdasarkan parameternya dari yang paling spesifik ke paling general (turunan ke base class). Jika tidak, maka pada saat kompilasi compiler C# akan menampilkan pesan kesalahan. Berikut adalah contoh penggunaan try dengan multiple catch block.

```
using System;
using System.IO;

namespace com.gotdotnet.otak
{
    class ContohMultipleCatch
    {
        public static void Main()
        {
            StreamReader sr = null;

            try
            {
                Console.WriteLine("Ketik nama file: ");
                string namaFile = Console.ReadLine();

                sr = File.OpenText(namaFile);
                string s = sr.ReadToEnd();
            }
        }
    }
}
```

```
        Console.WriteLine("**** Isi file {0}: ****",
namaFile);
        Console.WriteLine(s);
    }
    catch(FileNotFoundException)
    {
        Console.WriteLine("FileNotFoundException");
    }
    catch(Exception)
    {
        Console.WriteLine("Generik exception: {0}",
e.Message);
    }
    finally
    {
        if (sr != null)
            sr.Close();
    }
    Console.ReadLine();
}
}
```

6.5 User-Defined/Custom Exception

Untuk membuat exception sendiri, buat satu turunan dari Class Exception. Konvensi yang perlu diingat adalah:

Nama Class diakhiri dengan Exception, misalnya NamaTidakDikenalException.

Gunakan ApplicationException (atau turunannya) sebagai *base class*. Ini untuk membedakan dengan mudah antara exception yang dihasilkan oleh sistem dan aplikasi.

Class baru ini bisa diisi berbagai informasi yang mungkin diperlukan oleh kode client yang menangani exception ini.

Kapan kita perlu mendefinisikan exception sendiri? Gunakan custom exception bila exception yang diinginkan belum disediakan dalam framework (mungkin karena terlalu spesifik pada aplikasi) atau karena ada informasi tambahan yang ingin dimasukkan dalam obyek exception tsb. Contoh: bila Anda membangun aplikasi perbankan dan ingin mendefinisikan exception yang spesifik untuk dunia perbankan, misalnya "invalid account type", maka Anda perlu mendefinisikan exception baru.

6.6 Kapan Menggunakan Try Statement

Ada berbagai jenis error yang harus ditangani pembuatan aplikasi komputer. Salah satu pembagian yang umum adalah error yang bisa "diramalkan" kemunculannya dan *unexpected error* (tidak terduga). Termasuk dalam jenis pertama ini adalah *input error* (user memasukkan nilai input yang salah), pembagian dengan 0, dsb. *Unexpected error* adalah error-error yang tidak ada kaitannya dengan program itu sendiri, tapi berkaitan dengan pihak ketiga, misalnya lingkungan di mana program itu dijalankan. Termasuk dalam hal ini misalnya *out of memory*, *disk write error*, dkk. Batas antara kedua jenis error di atas tidak jelas. Yang ingin ditekankan dalam pembahasan ini adalah bukan klasifikasi error di atas, tapi mana jenis error yang cocok menggunakan try statement dan mana yang tidak, dipandang dari sisi performance aplikasi.

Kalau error tersebut sudah bisa diantisipasi di awal dan bisa ditangani dengan cara biasa, jangan gunakan try statement untuk menanganinya. Exception handling, dari sisi komputasinya, adalah sesuatu yang “mahal”. Gunakan cara biasa kalau memang itu error yang “biasa-biasa” saja. Contohnya: jika ada operasi pembagian, cek pembagiannya dengan cara biasa (if-then-else) Jangan mengandalkan exception untuk menangani error semacam itu. Itu ibaratnya menembak burung menggunakan rudal :p

```
try
{
    hasil = operand1 / operand2;
}
catch(Excepti on ex)
{
    Consol e. Wri teLi ne("Error: " + ex.getMessage());
}
-----
```

```
i f (operand2!=0)
{
    hasil = operand1 / operand2;
}
el se
{
    Consol e. Wri teLi ne("Error: pembagi an dengan 0");
}
```

6.7 Debugging

Tidak ada program yang 100% bug-free. Bagaimanapun kerasnya usaha kita untuk merancang dan membangun program yang benar, bug pasti akan muncul. VS.Net menyediakan fasilitas-fasilitas yang sangat memudahkan proses debugging. Jika Anda sebelumnya menggunakan VB6, fasilitas-fasilitas yang disediakan VS.Net tidak akan terasa asing lagi.

Melakukan debugging program sama ibaratnya dengan melakukan investigasi tindakan kriminal dengan berbekal hasil rekaman kamera tv security (CCTV). Untuk menentukan siapa yang menjadi tersangka, hasil rekaman tersebut perlu diputar (mungkin secara pelan-pelan) dan kemudian pada beberapa kondisi pemutaran tersebut dihentikan sementara (pause) agar kita bisa memperhatikan lebih seksama gambarnya. Dalam hal debugging, *bug* adalah si pelaku tindak kriminal, sedangkan *debugger* adalah video player yang digunakan untuk merunut kejadiannya. Si player itu sendiri (baca: VS.Net) punya banyak tombol yang masing-masing memiliki fungsi tersendiri. Sub-sub bab berikutnya memuat fungsi-fungsi yang paling sering digunakan dalam debugging sehari-hari.

6.7.1 Breakpoint

Jika video player punya tombol pause, maka VS.Net punya breakpoint. Fasilitas ini, seperti halnya tombol pause, digunakan untuk menghentikan sementara eksekusi program. Pada saat mendefinisikan breakpoint, kita bisa menentukan pada bagian mana dan kondisi apa eksekusi program akan dihentikan sementara. Hal ini terutama bermanfaat jika di awal kita sudah memiliki daftar “tersangka” (baris-baris kode yang mungkin menimbulkan terjadinya error).

Coba perhatikan contoh berikut:

```
using System;

namespace org.gotdotnet.otak
{
    class ContohBreakpoint
    {
        public static void Main()
        {
            Console.WriteLine("Masukkan nilai x : ");
            int x = int.Parse(Console.ReadLine());

            Console.WriteLine("Masukkan nilai y : ");
            int y = int.Parse(Console.ReadLine());

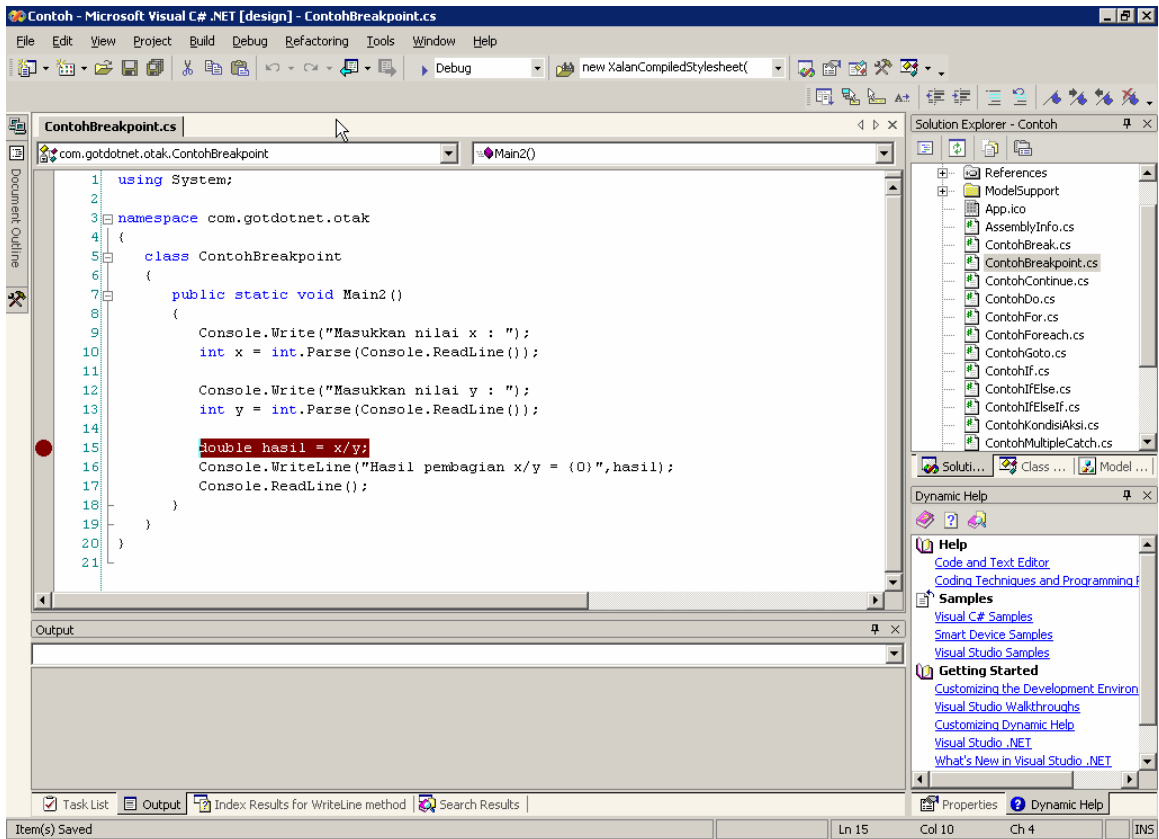
            double hasil = x/y;
            Console.WriteLine("Hasil pembagian x/y =
{0}", hasil);
            Console.ReadLine();
        }
    }
}
```

Program sederhana di atas berfungsi untuk menghitung hasil pembagian antara dua bilangan integer x dan y. Variabel hasil bertipe double, sesuai dengan kenyataan bahwa bila dua bilangan integer dibagi, maka hasilnya mungkin berbentuk pecahan.

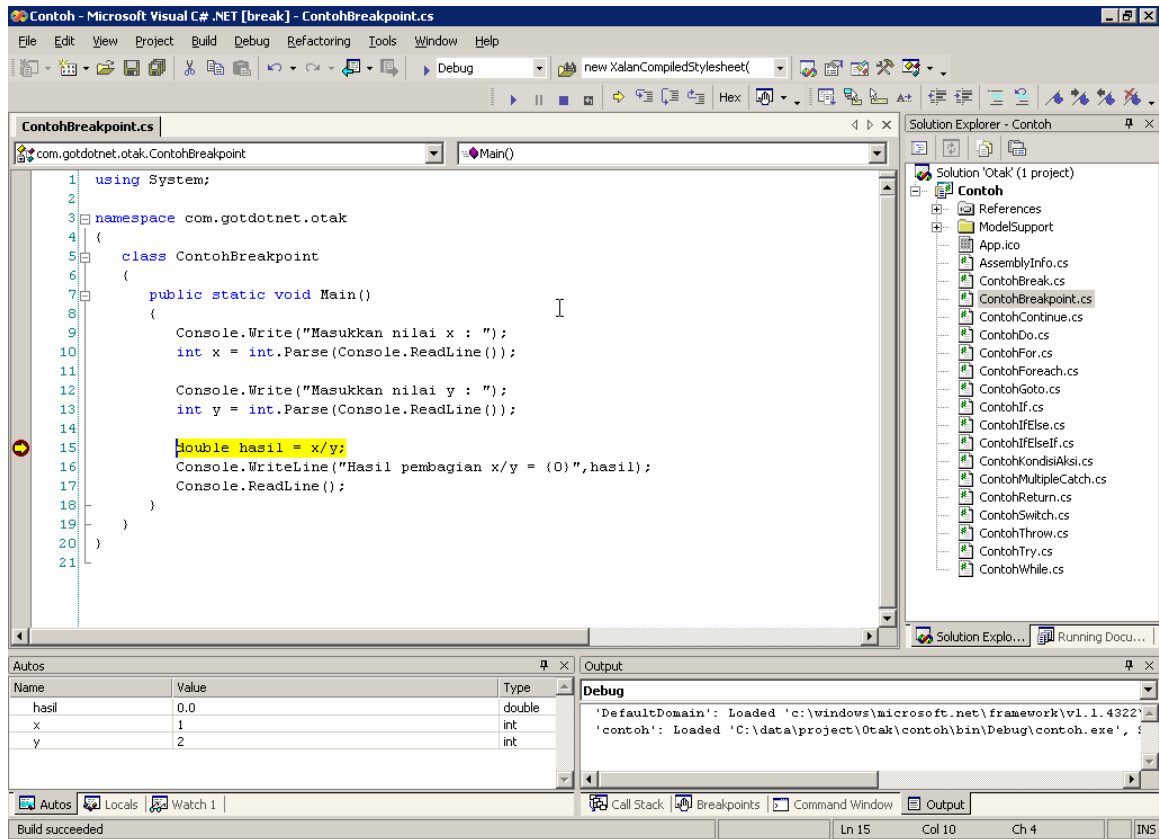
Walaupun sederhana, ternyata program tersebut menghasilkan nilai yang tidak terduga. Bila x=1 dan y=2, nilai yang ditampilkan ke layar adalah 0, bukan 0.5 seperti yang kita harapkan. Di mana kah letak kesalahannya?

Tanpa mengenal C# secara mendalam, kita bisa menebak bahwa kemungkinan penyebab kesalahannya ada dua, yaitu baris 15 (operasi pembagian) atau baris 16 (menampilkan hasil ke layar). Pada tahap ini, logika kita sudah benar, tapi implementasinya ternyata masih memiliki kesalahan. Inilah saatnya kita menggunakan fasilitas breakpoint yang disediakan VS.Net.

Ada beberapa cara untuk mengeset breakpoint. Cara ter gampang adalah dengan mengklik bagian berwarna abu-abu di sebelah kiri layar editor pada baris yang diinginkan. Bila breakpoint sudah diset (klik berhasil), akan muncul bulatan berwarna merah dan baris yang bersangkutan akan di-highlight kuning (lihat Gambar 6-2). Untuk menghapus breakpoint, klik bulatan berwarna merah tersebut atau pilih menu Debug | Clear All Breakpoints.



Gambar 6-2. Breakpoint – development mode



Gambar 6-3. Breakpoint – runtime mode

Perhatian: breakpoint baru akan berfungsi jika program dijalankan dalam debug mode. Pada release mode, setting breakpoint akan diabaikan oleh VS.Net.

Kembali pada program ContohBreakpoint, setelah selesai mengeset breakpoint, jalankan program dengan menekan tombol F5. Masukkan nilai x dan y sesuai permintaan program. Setelah nilai x dan y diperoleh, eksekusi program akan berhenti tepat pada baris dimana breakpoint diset (lihat Gambar 6-3 di atas). Pada saat ini, perintah-perintah sebelum baris tersebut sudah dieksekusi, sedangkan baris berwarna kuning itu sendiri **belum** dieksekusi. Jadi, pada saat suatu breakpoint ditemukan, eksekusi program akan dihentikan sementara agar kita bisa mengamati nilai berbagai variabel yang digunakan dalam program tsb. Cara mengamatinya diterangkan pada sub bab berikut: Auto, Locals, dan Watch.

6.7.2 Autos, Locals, dan Watch

Melanjutkan cerita di atas, pada saat eksekusi program dihentikan sementara, VS.Net secara otomatis akan menata ulang tampilannya. Pada saat tersebut, window autos pada bagian kiri bawah layar akan diaktifkan (Gambar 6-3). Perhatikan bahwa variabel x, y, dan hasil ditampilkan beserta nilainya. Jadi pada saat ini kita bisa mengamati situasi dan kondisi program yang nantinya mungkin menyebabkan error.

Pada bagian bawah window autos, ada beberapa tab lain, yaitu locals dan watch. Autos dan locals, jika coba-coba diklik, isinya sama. Window watch sendiri tidak ada isinya. Jadi apa perbedaan antara ketiga window ini?

Perbedaan antara autos, locals, dan watch dirangkum dalam tabel berikut.

Window	Keterangan
Autos	Diisi secara otomatis oleh VS.Net Isinya adalah variabel-variabel yang digunakan dalam statement yang sedang aktif dan statement sebelumnya
Locals	Diisi secara otomatis oleh VS.Net Isinya adalah variabel-variabel lokal
Watch	Tidak diisi secara otomatis oleh VS.Net. Kita harus mendaftarkan dulu apa yang hendak ditampilkan. Selain digunakan untuk mengamati variabel, watch bisa digunakan untuk mengamati nilai ekspresi. Contoh: selain utk mengamati nilai variabel x, watch bisa digunakan untuk mengamat nilai $x + y$

Untuk keperluan debugging masalah kita di atas, window autos sudah cukup memadai. Perhatikan bahwa nilai $x=1$, $y=2$, dan $hasil=0$ (nilai default karena baris sekarang belum dieksekusi). Untuk melengkapi persenjataan kita untuk debugging, kita masih memerlukan satu fitur lagi, yaitu *stepping*.

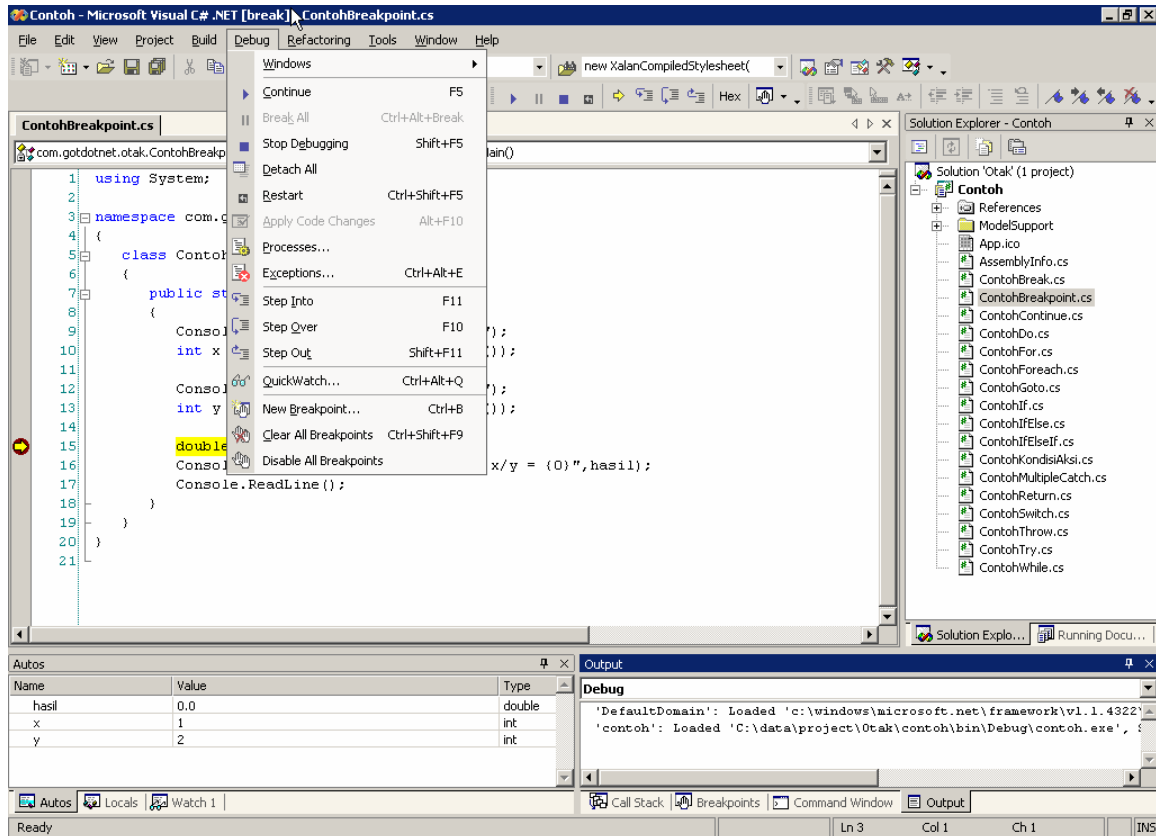
6.7.3 Step Into, Step Over, dan Step Out

Saat ini kita sudah bisa menghentikan program untuk sementara dan mengamati status/isi berbagai variabel. Ada satu lagi fitur penting untuk melakukan debugging, yaitu fitur untuk menjalankan program baris per baris, mulai dari pada saat breakpoint aktif. Dengan fitur ini, kita bisa memastikan baris mana yang menyebabkan terjadinya error.

Ada tiga perintah yang dapat digunakan untuk menjalankan program selangkah demi selangkah, yaitu Step Into, Step Over, dan Step Out. Untuk memanggil perintah-perintah tersebut, klik menu Debug dan pilih submenu ybs (lihat Gambar 6-4 di bawah). Fungsi perintah-perintah tersebut adalah sbb.

Nama Perintah	Kegunaan
Step Into	Digunakan untuk menjalankan baris berikutnya. Bila ada pemanggilan fungsi, maka program akan berhenti lagi pada baris pertama dalam fungsi itu. Jadi step into akan menyebabkan eksekusi program “masuk” ke ke dalam fungsi.
Step Over	Digunakan untuk menjalankan baris berikutnya, seperti halnya step into. Bedanya, kalo baris berikutnya memuat pemanggilan fungsi, step over akan menjalankan keseluruhan isi fungsi. Jadi apapun isi baris tersebut, semuanya akan dijalankan oleh step over sebelum eksekusi program di-pause lagi.
Step Out	Digunakan untuk keluar dari fungsi yang sedang

dijalankan. Misalnya debugging saat ini ada dalam fungsiB yang sebelumnya dipanggil oleh fungsiA. Jika step out dipanggil pada saat ini, maka sisa baris yang belum dijalankan dalam fungsiB akan dieksekusi, dan program akan berhenti pada baris berikutnya dalam fungsi pemanggil (fungsiA).

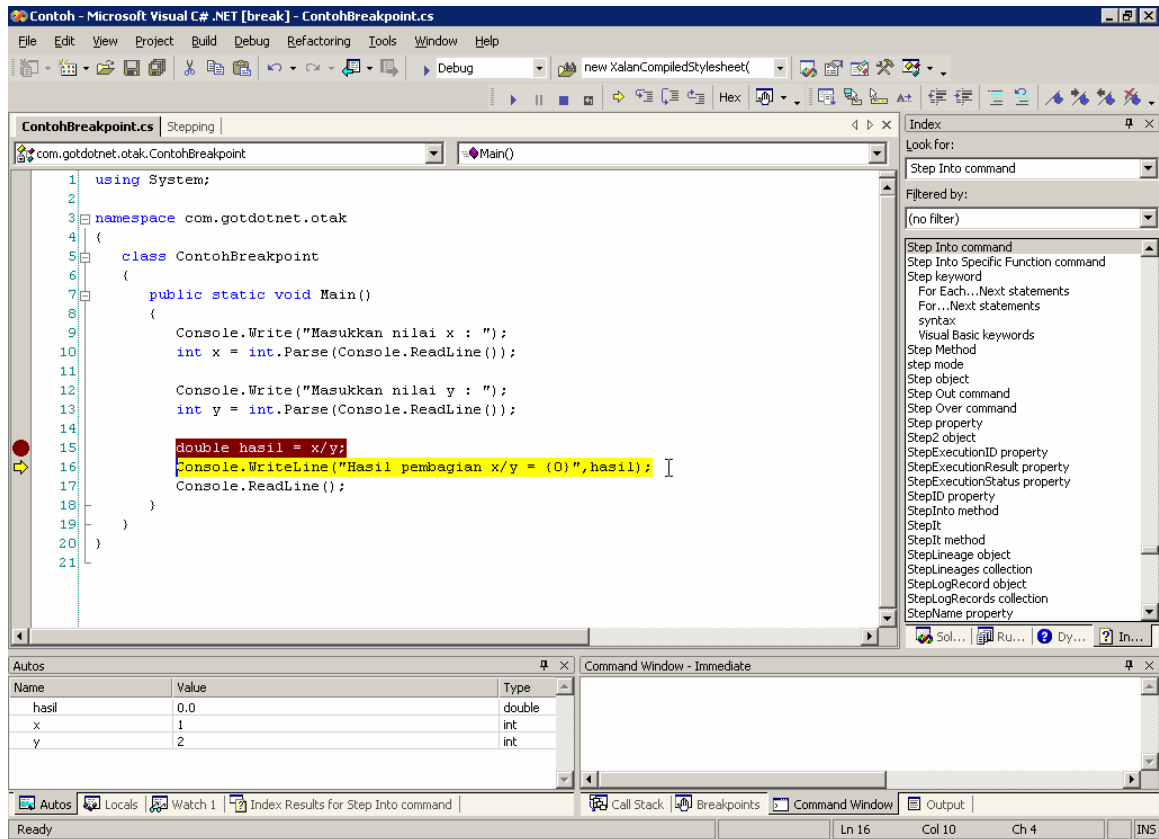


Gambar 6-4. Submenu step into, step over, dan step out.

Pada kasus yang sedang kita tangani, baris 15 dan 16 tidak memanggil fungsi lain, jadi menggunakan *step into* atau *step over* akan memperoleh hasil yang sama.

Sekarang coba tekan tombol F10 atau klik menu Debug | Step over. Selanjutnya yang akan di-*highlight* adalah baris 16 (lihat Gambar 6-5 di bawah). Ini berarti baris 15 sudah dijalankan dan variabel *hasil* seharusnya sudah memiliki nilai. Perhatikan lagi window autos di kiri bawah (masih pada Gambar 6-5).

Ternyata setelah baris 15 dijalankan, nilai variabel *hasil* masih 0. Dengan demikian pada titik ini kita telah mengetahui persis baris mana yang menyebabkan kesalahan. Jika perhitungannya benar, setelah baris 15 dijalankan variabel *hasil* seharusnya bernilai 0.5.



Gambar 6-5. Setelah step over dijalankan.

Selamat Anda sudah selangkah lebih maju dalam investigasi ini. Seringkali banyak waktu terbuang hanya untuk melokalisir masalah. Melokalisir masalah artinya memastikan bagian mana yang menyebabkan terjadinya kesalahan tsb. Jadi jika Anda membuat aplikasi untuk di-*maintain* sendiri, pastikan bahwa tiap kali terjadi kesalahan, ada cukup informasi yang dicatat (atau ditulis ke layar) sehingga Anda bisa tahu persis bagian mana yang menyebabkan terjadinya kesalahan tsb.

Jadi apa sebenarnya penyebab masalahnya? Kenapa hasil pembagiannya = 0, bukan 0.5? Operator pembagian bisa menerima operand bilangan bulat (misalnya integer) maupun pecahan (misalnya double). Dalam contoh tersebut, x dan y bertipe integer, sehingga operasi pembagian yang dilakukan adalah operasi pembagian bilangan integer, yang artinya bahwa nilai pecahan dari hasil operasinya akan dibuang. Jadi kalau yang dilakukan adalah x bagi y (1 bagi 2), hasilnya adalah 0. Kalau kita menginginkan hasilnya adalah 0.5, maka salah satu operand harus diubah ke double sehingga operasi yang dilakukan tidak lagi operasi pembagian bilangan integer. Contoh:

```

using System;

namespace com.gotdotnet.otak
{
    class ContohBreakpoint2
    {
        public static void Main()
        {
            Console.WriteLine("Masukkan nilai x : ");

```

```
int x = int.Parse(Console.ReadLine());
Console.WriteLine("Masukkan nilai y : ");
int y = int.Parse(Console.ReadLine());

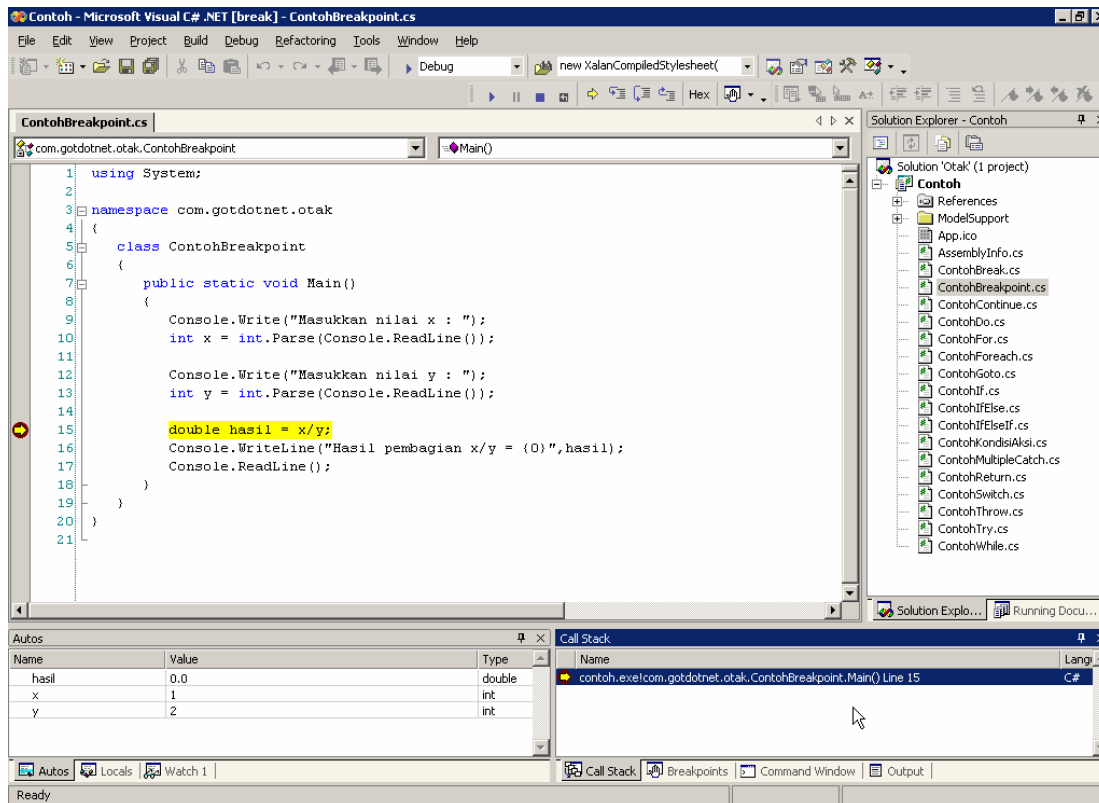
double hasil = (double)x/y;
Console.WriteLine("Hasil pembagian x/y =
{0}", hasil);
}
```

Pada Kode 7 di atas, pada bagian `double hasil = (double)x/y`, variabel `x` diubah tipenya ke `double` sebelum melakukan operasi pembagian. Dengan demikian yang dieksekusi adalah operasi pembagian untuk bilangan `double`.

Perhatikan juga pada contoh sebelumnya (Kode 6), walaupun tipe variabel hasil adalah `double`, yang terjadi adalah hasil pembagian tetap bertipe `integer` (bernilai 0) dan kemudian di-cast ke `double` (nilainya tetap 0). Jadi tipe variabel penampung hasil akhir operasi tidak berpengaruh dalam hal ini.

6.7.4 Call Stack

Window Call Stack terletak pada bagian kanan bawah layar, bersama-sama dengan breakpoint, command window, dan output window. Window ini digunakan untuk mengetahui urutan pemanggilan method yang sedang aktif saat ini. Misalkan pada saat breakpoint dicapai ada tiga fungsi yang dipanggil (nested): fungsiA memanggil fungsiB, dan fungsiB memanggil fungsiC yang memiliki breakpoint. Pada saat breakpoint tercapai, window call stack akan memuat ketiga fungsi tersebut dalam urutan sesuai dengan urutan pemanggilan,



Gambar 6-6. Call Stack

Fasilitas ini sangat berguna dalam melakukan debugging untuk suatu fungsi yang dipakai oleh banyak fungsi lain. Katakanlah kita membuat suatu fungsi *logError* dan fungsi ini dipanggil oleh banyak bagian dalam program yang sama. Jika suatu saat fungsi ini menghasilkan error tanpa tahu siapa yang memanggilnya, kita bisa menggunakan window call stack.

6.7.5 Command Window

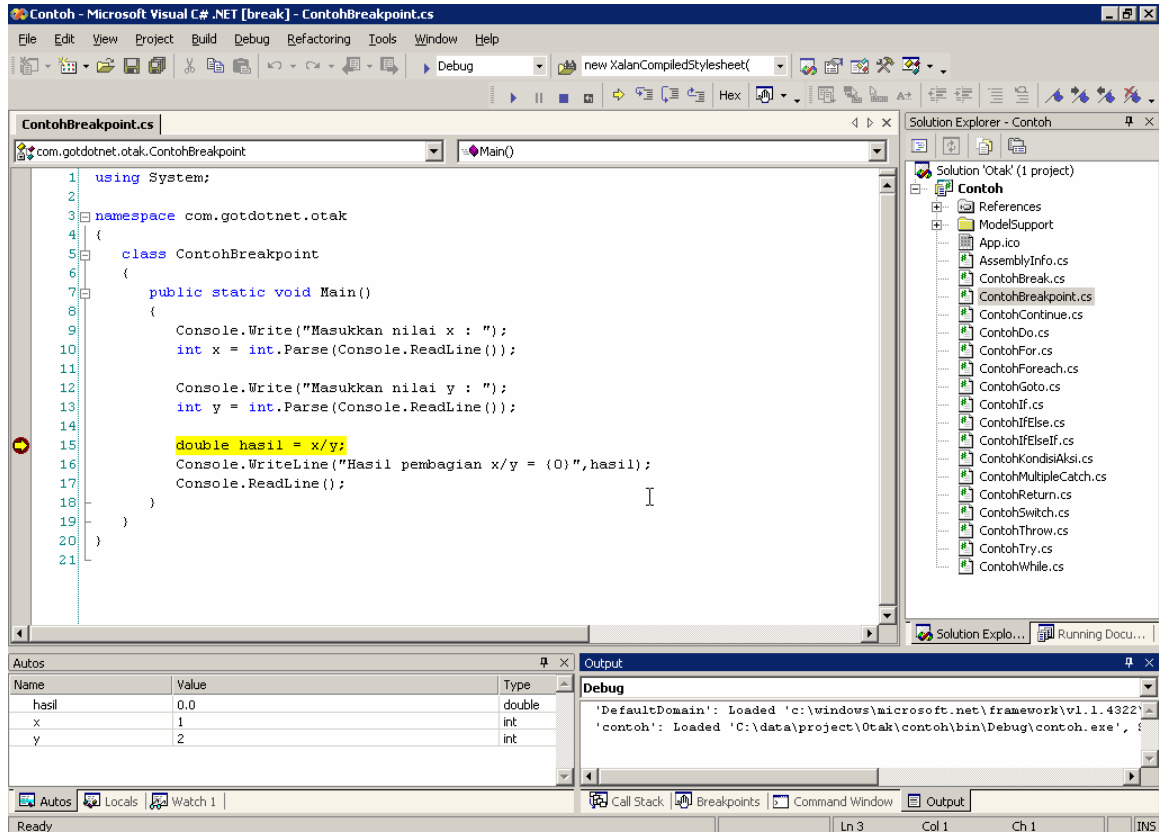
Command window memiliki dua mode yang masing-masing memiliki fungsi tersendiri. Kedua mode tersebut adalah mode *command* dan mode *immediate*. Dalam mode *command*, command window digunakan untuk menjalankan perintah-perintah VS.Net seperti new file, open file, dsb., tanpa mengakses sistem menu yang tersedia. Untuk tiap menu, VS.Net menyediakan perintah untuk dipanggil melalui command window, misalnya File.NewFile untuk membuat file baru, Debug.CallStack untuk menampilkan window Call Stack.

Para pengguna VB tentu sangat familiar dengan window immediate. Kegunaan mode immediate di VS.Net sama dengan yang tersedia di IDE VB6. Selain bisa digunakan untuk menjalankan perintah, dalam mode immediate kita bisa mengevaluasi ekspresi. Untuk mengevaluasi ekspresi, gunakan tanda tanya (?), misalnya `?x+y` akan menghasilkan nilai `x+y`. Untuk menjalankan perintah, gunakan tanda lebih besar (>), misalnya untuk membuka file baru, pada mode immediate gunakan perintah

>File | e. NewFile | e.

6.7.6 Output Window

Window output menampilkan pesan-pesan yang dihasilkan oleh VS.Net, misalnya pesan-kesalahan pada saat kompilasi. Aplikasi yang kita buat juga bisa memanfaatkan window ini untuk menampilkan pesan-pesan kesalahan. Gunakan Class Debug atau Trace untuk itu. Contoh keluaran pada output window ada pada gambar di bawah:



Gambar 6-7. Output Window

7. Pemrograman Object-Oriented

Norman Sasono

Pada bab-bab sebelumnya anda telah melihat syntax dari bahasa C#. Mulai dari bagaimana mendefinisikan variabel dan ekspresi, Flow Control, fungsi dan sebagainya. Pada bab ini, anda akan diajak untuk memahami prinsip dan teknologi yang melatar-belakangi bahasa C#, yaitu Object-Oriented Programming (OOP).

C# adalah sebuah bahasa pemrograman yang object-oriented. Dengan menggunakan bahasa pemrograman yang object-oriented, anda dapat membuat sebuah program dengan code yang berkualitas, mudah di maintain dan code yang dapat di re-use (dipakai di bagian lain dari program tanpa perlu anda meng-copy & paste baris-baris code tersebut).

Menulis program yang object-oriented sangat berbeda dengan menulis program yang bersifat prosedural. Untuk dapat menulis program yang object-oriented, anda harus memahami fitur-fitur object-oriented yang ada pada bahasa C#. Namun, untuk sampai ke sana, terlebih dahulu anda harus memahami dulu prinsip-prinsip dasar dari pemrograman object-oriented seperti misalnya: Abstraction, Encapsulation, Inheritance dan Polymorphisme. Bingung? Tidak usah. Lanjutkan saja membaca bab ini.

Dapat dipastikan, setelah anda memahami pemrograman object-oriented ini, anda akan mengerti seberapa powerful metodologi ini. Sekali anda terbiasa dengan pemrograman object-oriented, anda tidak akan mau untuk kembali ke pemrograman bergaya prosedural lagi.

7.1 Object

Pada sub-bab ini akan dijelaskan tentang apa sebenarnya sebuah object. Disini, penjelasan akan dibagi-bagi untuk topik-topik berikut:

- Apa itu Object?
- Definisi formal
- State
- Behavior
- Identitas

7.1.1 Apa itu Object?

Mudahnya, sebuah object adalah sebuah benda. Benda apa saja yang dapat anda kenali atau bayangkan. Nyata maupun abstrak. Fisik maupun konseptual. Termasuk juga software. Contoh benda fisik adalah mobil, truk, rumah, buku, dokumen, harddisk, printer, keyboard dan lain-lain. Contoh benda konseptual misalnya sebuah proses kimia.

Tetapi, definisi object tidak berhenti disitu saja. Bagi programmer, object adalah sebuah benda yang dapat anda nyatakan (represent) dalam sebuah program. Seorang customer adalah sebuah object yang dapat anda nyatakan dalam program. Demikian juga produk, perusahaan, hingga benda-benda seperti database, file, dan lain-lain.

Alasan mengapa kita menyatakan object-object tersebut di dalam sebuah program adalah untuk memungkinkan kita menulis code yang memodelkan dunia nyata ke dalam program, dan memungkinkan kita untuk memecah program menjadi unit-unit kecil yang tentunya lebih mudah di bangun dan di manage. Dan tentu saja, memungkinkan adanya code re-use.

7.1.2 Definisi Formal

Secara formal, object didefinisikan sebagai sebuah benda (entity) yang memiliki batasan (boundary) dan identitas (identity) yang terdefinisi dengan jelas, yang membungkus (encapsulate) kondisi (state) dan perilaku (behavior).

Berdasarkan definisi diatas, kita dapat menyatakan bahwa:

- Sebuah object adalah sebuah benda yang memiliki batasan yang terdefinisi dengan jelas. Maksudnya, tujuan dari object tersebut harus jelas.
- Sebuah object memiliki dua hal: kondisi (state) dan perilaku (behavior).
- Kondisi (state) dari suatu object sering dinyatakan melalui attribute dari obyek tersebut. Sedangkan perilaku (behavior) dinyatakan melalui operations dari object tersebut.
- Pada sub bab-sub bab selanjutnya, akan dibahas lebih jauh tentang state dan behavior dari suatu object. Demikian juga dengan attribute dan operations.

7.1.3 State

Setiap object memiliki state. State dari suatu object adalah satu dari sekumpulan kondisi yang mungkin di mana object tersebut berada. Biasanya, dengan berjalannya waktu, state dari sebuah object selalu berubah.

State dari sebuah object dinyatakan sebagai sekelompok properti yang disebut attribute, berikut juga nilai dari setiap properti ini.

Bingung? Mungkin lebih mudah jika digunakan sebuah contoh. Sebuah mobil adalah sebuah object. Nah, mobil ini memiliki beberapa properti seperti warna, jumlah roda, kapasitas mesin, jumlah pintu dan lain-lain. Setiap properti ini memiliki nilai.

Misal:

- Warna mobil adalah hijau
- Jumlah Roda adalah 4
- Kapasitas Mesin adalah 2000 cc
- Jumlah Pintu adalah 2

- Hijau, 4, 2000 cc dan 2 adalah nilai dari properti / attribute mobil tersebut.
- Contoh lain, seorang employee adalah sebuah object. Nama employee, tanggal lahir dan jabatan adalah properti atau attribute dari employee. Lalu, "Agus", "13 Agustus 1960", "CEO" adalah nilai untuk properti-properti tadi.
- Sebuah file di komputer juga sebuah object. File memiliki nama, size, tipe, path dan lain-lain sebagai properti. Lalu "Sales Report.doc", "20 Mega Byte", "Word Document" dan "C:\MyDocuments" adalah nilai untuk properti-properti tadi.

Melalui contoh diatas, diharapkan cukup jelas bagi anda apa itu object, apa itu properti atau attribute dari object termasuk juga bahwa properti tersebut memiliki suatu nilai.

Lalu bagaimana dengan State? State suatu object tidak dinyatakan melalui suatu kondisi /nilai sebuah attribute atau sekumpulan attribute saja. Melainkan, state didefinisikan oleh total dari seluruh attribute yang dimiliki object tersebut. Contohnya: Jika Profesor Agus adalah seorang dosen dengan status dosen kontrak, lalu statusnya berubah menjadi dosen tetap, maka state dari object Agus ini dinyatakan berubah. Bukan hanya attribute status-nya saja.

Berikut adalah contoh yang lebih lengkap tentang state dari object Profesor Agus yang dinyatakan dalam attribute-attribute:

Nama: Agus
NIP: 007123
TanggalMasuk: 01/01/2004
Status: Dosen Tetap
Spesialisasi: Network Programming
JadwalMaximum: 3

7.1.4 Behavior

Behavior adalah perilaku. Perilaku bagaimana sebuah object ber-aksi dan ber-reaksi. Sebuah object akan melakukan aksi atau reaksi terhadap request atau permintaan dari object lain.

Behavior sebuah object dinyatakan dengan operasi-operasi yang dapat dilakukan oleh object tersebut. Contoh: Profesor Agus dapat melakukan aksi seperti mengambil cuti misalnya. Behavior ini dinyatakan dengan operasi Cuti().

Contoh lain, Profesor Agus dapat memeriksa nilai mahasiswa dan menyimpan nilai tersebut. Menyimpan nilai mahasiswa tadi dapat dinyatakan dengan operasi SimpanNilai().

7.1.5 Identitas

Setiap object memiliki identitas yang unik. Meskipun state dari dua object dapat saja sama, tetapi dua object tersebut tetap memiliki identitas yang unik.

Di dalam dunia nyata, dua orang yang berbeda dapat saja memiliki karakteristik yang sama. Nama yang sama, tanggal lahir yang sama, pekerjaan yang sama, gaji yang sama. Tetapi, tetap saja dua orang tersebut adalah dua orang yang berbeda. Masing-masing memiliki identitas unik-nya sendiri.

Demikian pula halnya konsep object pada software. Dua object yang sama dapat saja memiliki state yang sama (attribute), tetapi kedua object tersebut object yang masing-masing memiliki identitas unik mereka sendiri.

7.2 Prinsip Dasar Object-Oriented

Setelah anda memahami apa yang dimaksud dengan object, state dan behavior, sekarang anda akan diajak untuk melihat prinsip-prinsip yang mendasari konsep object-oriented.

Ada 4 hal:

- Abstraction
- Encapsulation
- Modularity
- Hirarki

7.2.1 Abstraction

Apa itu abstraction? Abstraction adalah karakteristik mendasar yang dimiliki oleh sebuah entity (benda) yang membedakan entity tersebut dari semua jenis entity yang lain. Dengan abstraction, kita dapat mengurangi kompleksitas dengan berkonsentrasi pada karakteristik mendasar dari sebuah entity tersebut.

Akan tetapi, abstraction itu bergantung pada domain atau perspective. Artinya, hal-hal yang penting dalam suatu konteks dapat saja tidak penting dalam konteks yang lain.

Harus diakui, memang tidak mudah untuk menjelaskan apa yang dimaksud dengan abstraction. Abstraction bersifat sangat konseptual. Berikut ini diberikan beberapa contoh abstraction:

Seorang mahasiswa adalah seorang murid di sebuah universitas.

Seorang dosen adalah seorang pengajar di suatu universitas.

Sebuah mata kuliah adalah pelajaran atau subyek yang diajarkan di suatu universitas.

Seorang customer adalah seorang pembeli dari suatu produk.

Mahasiswa, dosen, mata kuliah dan customer adalah abstraction dari sebuah konsep. Dari contoh-contoh ini, diharapkan anda dapat memahami apa itu abstraction.

7.2.2 Encapsulation

Encapsulation sering juga dinyatakan sebagai penyembunyian informasi. Encapsulation memungkinkan seseorang untuk melakukan sesuatu tanpa perlu tahu secara mendetail tentang bagaimana sesuatu tersebut dilakukan.

Contoh yang paling mudah adalah penggunaan remote control TV. Jika anda (client) ingin memindahkan saluran TV dari satu saluran ke saluran lain, anda cukup menekan tombol saluran yang anda kehendaki pada remote control anda. Itu saja. Anda tidak perlu tahu secara mendetail implementasi dari serangkaian mekanisme elektronik yang kompleks yang terjadi di dalam remote control dan TV anda. Anda cukup tahu bagaimana menekan tombol remote control anda.

Contoh lain, jika anda menginjak rem mobil anda, anda tidak perlu tahu implementasi dari mekanisme mekanik bagaimana mobil anda itu berhenti. Anda cukup tahu bagaimana menginjak rem mobil anda.

Itulah encapsulation. Menyembunyikan semua detail informasi dan mekanisme.

Encapsulation menghilangkan ketergantungan pada implementasi. Sehingga, dimungkinkan untuk mengganti implementasi tanpa merubah pengetahuan client. Pada contoh diatas, implementasi dari mekanisme elektronik pemindahan saluran pada remote control dan TV anda dapat saja diganti/diubah, tetapi bagi anda (client) tetap saja anda cukup tahu bagaimana menekan tombol remote control.

Encapsulation juga melindungi informasi internal dari suatu object. Client tidak dapat mengakses informasi internal tersebut. Client hanya dapat meminta suatu object untuk melakukan suatu operasi yang merubah informasi internal tadi.

7.2.3 Modularity

Modularity adalah memecah-mecah sesuatu yang kompleks menjadi bagian-bagian kecil yang lebih mudah di manage. Modularity memudahkan dalam memahami sebuah system yang kompleks.

Suatu system software yang kompleks dapat dipecah-pecah menjadi bagian-bagian kecil yang lebih sederhana. Bagian-bagian kecil ini dapat dibangun satu per satu secara independen sepanjang interaksi antar bagian kecil ini dimengerti.

Contohnya, sebuah system software pembelian barang melalui internet. System ini sedemikian besar dan kompleks untuk dimengerti. Maka itu, system ini dipecah-pecah menjadi modul-modul berikut:

- System pemesanan.
- System pembayaran.
- System pengiriman.

Modul-modul diatas dapat dikembangkan secara independen dan ketiganya adalah building block dari system yang lebih besar yaitu system pembelian barang melalui internet.

7.2.4 Hirarki

Hirarki adalah pengurutan dari suatu abstraction menjadi sebuah struktur pohon. Hirarki adalah sebuah organisasi taxonomi.

Dengan menggunakan hirarki, memudahkan untuk mengenali persamaan dan perbedaan dari anggota dalam hirarki tersebut. Suatu contoh, misalnya botani. Botani mengelompokkan tumbuhan menjadi beberapa keluarga tumbuhan.

7.3 Object dan Class

Setelah anda memahami konsep tentang object-oriented, selanjutnya akan dibahas bagaimana merepresentasikan konsep tentang object yang telah dielaskan sebelumnya ke dalam software atau program anda.

Pada pemrograman object-oriented, object-object diimplementasikan melalui suatu class. Bab selanjutnya didedikasikan khusus untuk membahas mengenai class secara mendetail. Pada bab ini, hanya akan diberikan pengantar mengenai class tersebut, meliputi:

- Class
- Attribute
- Operation
- Polymorphism
- Inheritance

7.3.1 Class

Apa itu class? Sebuah class adalah sebuah deskripsi dari sekumpulan object yang memiliki atribut, operasi dan hubungan antar object yang sama. Sebuah object dinyatakan sebagai sebuah instance dari sebuah class. Sebuah class adalah blueprint dari sebuah object.

Contoh yang paling sering dipakai adalah cetakan kue agar-agar. Cetakan agar-agar tadi sering di analogikan sebagai sebuah class. Sedangkan kue agar-agar-nya adalah object/instance dari cetakan tadi. Jadi, cetakan agar-agar adalah bukan sebuah kue agar-agar. Demikian juga class. Class bukanlah sebuah object. Class adalah blueprint yang membentuk sebuah object, dan sebuah object adalah instance dari sebuah class.

Secara grafis, sebuah class dapat digambarkan menjadi sebuah segiempat yang terbagi menjadi beberapa kompartemen seperti ini:



Gambar 7-1 Class Dosen

Jadi, sebuah class adalah sebuah definisi abstract dari sebuah object. Class mendefinisikan struktur dan behavior dari setiap object class tersebut. Class berfungsi sebagai template untuk membuat object.

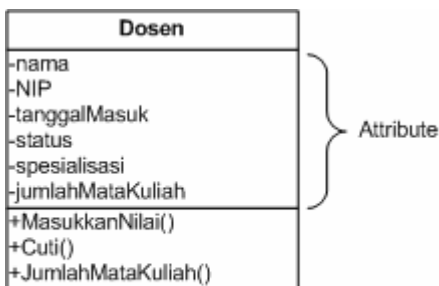
7.3.2 Attribute

Attribut adalah properti dari sebuah class yang memiliki nama dan mendeskripsikan suatu range nilai dari properti tersebut.

Sebuah class dapat memiliki attribut, tetapi dapat juga tidak. Pada suatu waktu, sebuah object dari sebuah class akan memiliki nilai spesifik untuk setiap attribut class tersebut.

Setiap attribut memiliki apa yang disebut Type, yang menyatakan attribut jenis apakah dia. Type yang umum untuk suatu attribut adalah integer, boolean, real atau enumeration. Type-type ini sering disebut juga Type yang primitive. C#, seperti halnya bahasa pemrograman yang lain juga memiliki type primitive yang spesifik.

Secara grafis, kompartemen yang berada di bawah nama class adalah *tempat attribut-attribut* dari sebuah class:



Gambar 7-2 Attribute sebuah class

7.3.3 Operation

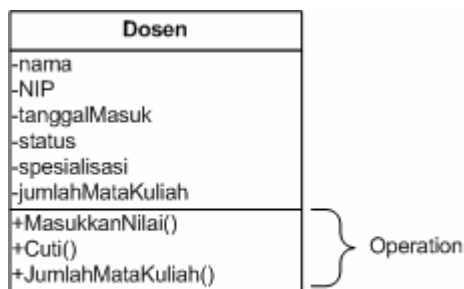
Operation adalah implementasi dari sebuah service yang dapat diminta oleh object-object lain dari class untuk mempengaruhi behavior-nya. Mudah-mudahan, operation dari sebuah class mendeskripsikan apa yang dapat dilakukan oleh class tersebut.

Operation dapat berbentuk perintah (command) ataupun pertanyaan (question). Melalui operation berbentuk perintah ini state ataupun juga nilai dari sebuah attribute dapat diubah.

Sebuah operation dideskripsikan dengan sebuah return type, nama operation dan dengan nol (0) atau lebih parameter. Secara keseluruhan, return type, nama dan parameter dari sebuah operation disebut sebagai signature dari operation tersebut.

Sebuah class boleh memiliki sejumlah operation atau tidak sama sekali.

Secara grafis, operation digambarkan pada kompartemen paling bawah dari sebuah class:



Gambar 7-3 Operation dari sebuah class

7.3.4 Polymorphism

Polymorphism adalah kemampuan untuk menyembunyikan berbagai implementasi yang berbeda dibelakang sebuah interface.

Kita kembali ke contoh tentang remote control dan TV tadi. Sebuah remote control, dapat digunakan untuk beberapa jenis televisi (implementation) yang mendukung interface spesifik bagaimana remote control tersebut didesain.

Sebagai contoh Polymorphism di program anda, misalnya sebuah object ingin tahu luas permukaan dari beberapa bentuk matematis sederhana. Misalnya, segi empat, segi tiga dan lingkaran. Luas permukaan untuk tiap-tiap bentuk tadi dihitung melalui rumus yang sangat berbeda:

$$\begin{aligned} \text{Luas Segi Empat} &= \text{sisi} \times \text{sisi} \\ \text{Luas Segi Tiga} &= \frac{1}{2} \times \text{alas} \times \text{tinggi} \\ \text{Luas Lingkaran} &= \text{pi} \times \text{jari} \times \text{jari} \end{aligned}$$

Jika tidak menggunakan Object-Oriented dalam memprogram, program anda akan menjadi seperti ini:

```
if (bentuk == "Segi Empat")
{
    HitungLuasSegiEmpat();
}
if (bentuk == "Segi Tiga")
{
    HitungLuasSegiTiga();
}
if (bentuk == "Lingkaran")
{
    HitungLuasLingkaran();
}
```

Dengan teknologi object, setiap bentuk matematis tadi dapat direpresentasikan menjadi sebuah class dan setiap class tahu bagaimana menghitung luas permukaannya sendiri melalui satu operasi "HitungLuas()". Object yang meminta nilai luas permukaan tiap-tiap bentuk matematis (misal, Lingkaran) cukup meminta object lingkaran tadi untuk menghitung luas permukaannya. Object yang meminta nilai luas tadi tidak perlu mengingat signature yang berbeda-beda untuk tiap bentuk matematis. Object ini cukup memanggil operasi "HitungLuas()" yang dimiliki oleh tiap-tiap bentuk matematis tadi.

Polymorphism memungkinkan pesan yang sama (dalam contoh ini: HitungLuas()) untuk ditangani dengan cara yang berbeda tergantung object yang menerima permintaan tadi. Lingkaran akan mengerjakan HitungLuas() dengan cara yang berbeda dengan Segi Tiga ataupun Segi Empat. Akan tetapi, bagi object client (object yang meminta nilai luas), cukup tahu operasi HitungLuas().

7.3.5 Inheritance

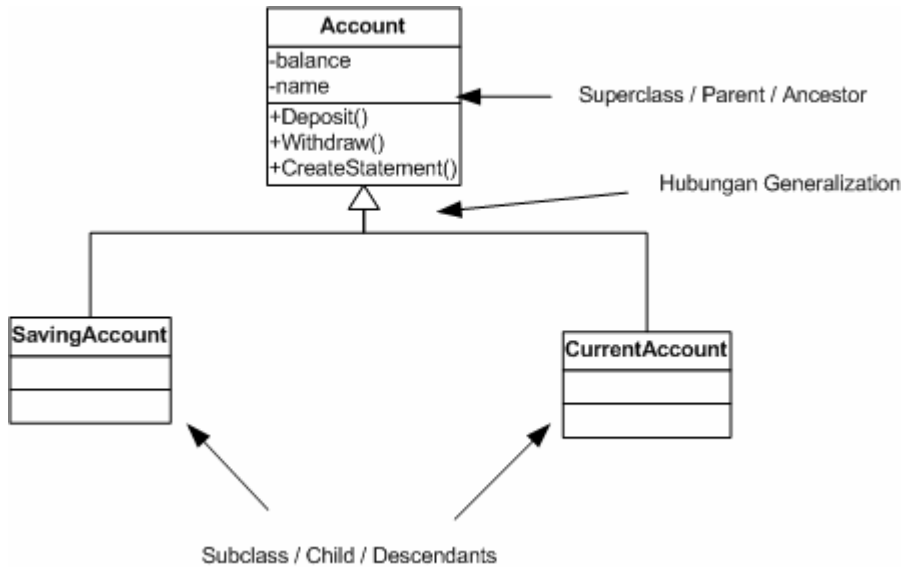
Sebelum melihat apa itu Inheritance, kita lihat dulu suatu konsep yaitu Generalization.

Generalization adalah hubungan antar class dimana satu class mengambil struktur dan behavior yang sama dari suatu class lain atau lebih. Generalization mendefinisikan hirarki dari suatu abstraction dimana sebuah subclass mewarisi (inherit) dari sebuah superclass atau lebih.

Generalization dapat dinyatakan sebagai hubungan "adalah". Misal, singa adalah hewan, rumput adalah tumbuhan, dan sebagainya. Di contoh ini, hewan adalah generalization dari singa dan tumbuhan adalah generalization dari rumput. Sebaliknya, singa inherits dari hewan dan rumput inherits dari tumbuhan.

Terdapat dua macam Inheritance, yaitu single inheritance dan multiple inheritance.

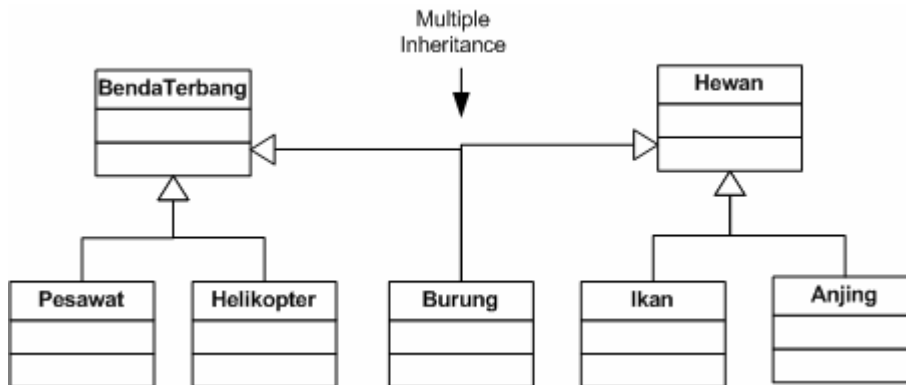
Untuk lebih jelas, berikut adalah contoh dari single Inheritance:



Gambar 7-4 Single Inheritance

Generalization digambarkan sebagai hubungan dari subclass ke superclass. Sedangkan Inheritance adalah pewarisan struktur dan behavior dari superclass ke subclass. Dalam contoh diatas, class-class SavingAccount dan CurrentAccount mewarisi operasi-operasi class Account seperti Deosit(), Withdraw() dan CreateStatement().

Suatu contoh dari suatu multiple inheritance dapat digambarkan sebagai berikut:



Gambar 7-5 Multiple Inheritance

Multiple Inheritance maksudnya adalah sebuah class dapat inherit dari beberapa class (lebih dari satu). Sebagai contoh, pada gambar diatas, seekor burung inherit dari dua class: BendaTerbang dan Hewan. Class Burung akan memiliki struktur dan behavior dari class BendaTerbang maupun Hewan.

Tidak ada yang salah pada multiple inheritance, malah mungkin dibutuhkan untuk memodelkan suatu konsep dunia nyata ke dalam program dengan lebih akurat. Akan tetapi, kita harus berhati-hati dalam menggunakan multiple inheritance ini, jika tidak menggunakannya dengan cermat, multiple inheritance malah dapat meningkatkan kompleksitas masalah yang hendak kita pecahkan. Bukannya menyederhanakan.

Disamping itu, tidak seluruh bahasa pemrograman mendukung konsep multiple inheritance ini. Contohnya, C#. **C# tidak mendukung multiple inheritance. Anda tidak dapat menggunakan multiple inheritance dengan C#.**

Pada umumnya, sebuah class hanya inherit dari satu class saja.

Lalu, apa saja yang di warisi (inherit) oleh sebuah subclass (child) dari superclass-nya (parent)?

Subclass meng-inherit attribut dari parent, operation dan hubungan parent terhadap class-class lain.

Subclass boleh memiliki attribut, operasi dan hubungan tambahan.

Subclass juga boleh mendefinisi ulang operation yang diwarisi dari parent.

Attribut, operasi dan hubungan yang umum biasanya berada pada class dengan level tertinggi pada hirarki.

7.4 Object-Oriented C#

Untuk lebih memahami konsep-konsep OOP ini akan lebih mudah jika digunakan sebuah contoh program OOP in action. Melalui contoh dan baris-baris program berikut, diharapkan anda dapat lebih memahami OOP.

Kita akan kembali menggunakan contoh class Account atau rekening bank. Class account ini adalah merupakan sebuah abstraction dari sebuah konsep di dunia perbankan yaitu rekening bank. Class Account dapat digambarkan sebagai berikut:

Account
-balance : float
+Account() +Deposit(in amount : float) +Withdraw(in amount : float) +TransferFunds(in destination : Account, in amount : float) +Balance()

Gambar 7-6 Account Class

Dari gambar diatas, kita dapat melihat bahwa class Account mempunyai:

1 Attribut: balance, dengan Type float

5 Operation:

- Account() – merupakan constructor
- Deposit() – menerima sebuah parameter masuk bernama amount & ber-Type float
- Withdraw() - menerima sebuah parameter masuk bernama amount & ber-Type float
- TransferFunds() – menerima dua parameter. Parameter bernama destination dengan Type Account, dan parameter bernama amount dengan Type float.
- Balance()

Sedangkan jika class Account diatas kita wujudkan dalam code C# akan menjadi sebagai berikut:

```
using System;
namespace Bank
{
    public class Account
    {
        public Account()
        {
        }

        private float balance;

        public float Balance
        {
            get
            {
                return balance;
            }
        }

        public void Deposit(float amount)
        {
            balance += amount;
        }

        public void Withdraw(float amount)
        {
            balance -= amount;
        }

        public void
        TransferFunds(Account destination, float amount)
        {
            destination.Deposit(amount);
            this.Withdraw(amount);
        }
    }
}
```

Sekarang, kita akan meninjau program diatas baris demi baris.

```
public Account()
{
}
```

Baris diatas disebut constructor dari sebuah class, maksudnya, baris-baris yang ada dalam operasi ini akan dijalankan saat ada object client yang meng-instantiate class Account ini. Jika di dalam constructor tidak ada code yang dijalankan, artinya constructor dari class Account ini tidak melakukan apa-apa. Perlu diperhatikan bahwa nama dari constructor adalah sama dengan nama dari class.

```
private float balance;
```

Baris diatas diatas mendefinisikan sebuah attribute dari class account yaitu balance. Adapun Type dari balance ini adalah float. Balance adalah saldo dari rekening bank. Saldo suatu rekening dapat bertambah maupun berkurang. Akan tetapi, kita tidak dapat secara langsung menambah atau mengurangi nilai balance tadi. Ini yang disebut dengan encapsulation. Nilai balance ter-encapsulate dan hanya diubah nilainya melalui suatu operasi yang dimiliki class.

Untuk menambah nilai saldo/balance, kita menggunakan sebuah operasi penyetoran, yaitu operasi Deposit() berikut:

```
public void Deposit(float amount)
{
    balance += amount;
}
```

Sedangkan untuk mengurangi saldo, dilakukan melalui operasi penarikan, yaitu operasi Withdraw() berikut:

```
public void Withdraw(float amount)
{
    balance -= amount;
}
```

Untuk kedua operasi ini, terdapat parameter yaitu amount yang ber-Type float.

Operasi lain yang dapat dilakukan adalah transfer uang dari satu account ke account lain. Hal ini dilakukan dengan operasi TransferFunds() berikut:

```
public void TransferFunds(Account destination, float amount)
{
    destination.Deposit(amount);
    this.Withdraw(amount);
}
```

Operasi ini menerima dua parameter. Destination yang ber-Type account dan amount yang ber-Type float. Di dalam operasi ini, dua operasi dilakukan. Operasi Deposit() terhadap object destination yang merupakan parameter operasi dan operasi Withdraw() yang dilakukan terhadap Account sendiri.

Jika operasi yang memiliki nama yang sama dengan nama class disebut constructor, di dalam C#, operasi-operasi seperti Deposit(), Withdraw() dan TransferFunds() disebut dengan Method.

Selanjutnya, kita akan melihat operasi khusus berikut:

```
public float Balance
{
    get
    {
        return balance;
    }
}
```

Operasi ini digunakan untuk mengakses (membaca) nilai dari balance pada suatu saat. Operasi khusus semacam ini di dalam C# disebut dengan Property.

Jadi, anda sudah diperkenalkan dengan apa yang disebut constructor, method dan property.

Untuk melihat bagaimana class Account ini digunakan dalam program, kita dapat menggunakan sebuah program console bernama Class1 yang akan bertindak sebagai client object.

Berikut ini adalah program console tersebut:

```
using System;
namespace Bank
{
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            Account source = new Account();
            source.Deposit(200.00F);
            Console.WriteLine
                ("source initial balance : " +
                 source.Balance.ToString());

            Account destination = new Account();
            destination.Deposit(150.00F);
            Console.WriteLine
                ("destination initial balance : " +
                 destination.Balance.ToString());

            source.TransferFunds(destination, 100.00F);
            Console.WriteLine
                ("source final balance : " +
                 source.Balance.ToString());
            Console.WriteLine
                ("destination final balance : " +
                 destination.Balance.ToString());
        }
    }
}
```

Kita akan melihat baris demi baris program console ini secara kronologis. Perhatikan baris-baris program didalam operasi Main().

```
Account source = new Account();
source.Deposit(200.00F);
Console.WriteLine
    ("source initial balance : " +
     source.Balance.ToString());
```

Disini, console meng-instantiate class Account menjadi object dengan nama source. Selanjutnya dipanggil operasi Deposit(), dimana dimasukkan sebesar 200.00F ke object source tadi. Selanjutnya ditampilkan di console nilai dari balance object source.

```
Account destination = new Account();
destination.Deposit(150.00F);
Console.WriteLine
    ("destination initial balance : " +
     destination.Balance.ToString());
```

Kali ini, class account di-instantiate menjadi object dengan nama destination. Operasi Deposit() dilakukan untuk menambah balance dari destination menjadi 150.00F. Selanjutnya, ditampilkan juga balance dari destination di console.

```
source.TransferFunds(destination, 100.00F);
Console.WriteLine
```

```
        ("source final balance : " +  
         source.Balance.ToString());  
Console.WriteLine  
        ("destination final balance : " +  
         destination.Balance.ToString());
```

Operasi `TransferFunds()` dari object `source` dipanggil. Operasi ini akan memindahkan uang dari `source` ke object `destination` yang juga ber-Type `account` yang dilempar melalui parameter operasi. Nilai `amount` yang akan ditransfer juga menjadi parameter operasi, yaitu float sebesar 100.00F. Setelah itu, nilai `balance` dari `source` dan `destination` sesudah proses transfer juga ditampilkan di console.

Setelah pemanggilan method `TransferFunds()` ini, `balance` dari `source` akan berkurang 100.00F dan `balance` dari `destination` akan bertambah 100.00F.

Berikut adalah hasil jika program console `Class1` dijalankan:

```
source initial balance : 200  
destination initial balance : 150  
source final balance : 100  
destination final balance : 250
```

Contoh ini adalah contoh yang sangat sederhana dan tidak mewakili program yang dipakai di dunia perbankan sesungguhnya. Akan tetapi, dari contoh ini, diharapkan anda dapat melihat bagaimana sebuah program OOP in action. Setelah anda melihat contoh program pasti anda memiliki pemahaman yang lebih baik dari konsep-konsep OOP yang telah dijelaskan sebelumnya. Tidak ada salahnya anda membaca-baca kembali konsep-konsep OOP tersebut. Terutama pada topik-topik seperti: *abstraction*, *encapsulation*, *polymorphism* dan *inheritance*.

Lanjutkanlah membaca bab-bab selanjutnya dari buku ini setelah anda benar-benar paham apa itu OOP (Object-Oriented Programming).

8. Class

Fathur Rahman

Class! Nanti anda akan tahu, dalam C#, semuanya dibangun diatas Class. Tidak ada C# tanpa Class. Apakah Class itu?

Awalnya saya memahami istilah ini sama seperti Class-Class di sekolah-sekolah, atau klasifikasi dalam pelajaran biologi, intinya semua yang berhubungan dengan golong mengolong. Samakah? Ini tidak sama. Jika pemahaman anda sama dengan saya, maka saya menyarankan anda membuang jauh-jauh pengertian itu dalam konteks ini.

Lantas Class dalam konteks ini apa maksudnya? Definisi Class menurut standart ECMA adalah sebagai berikut:

“Class adalah struktur data yang berisi data-data (konstanta-konstanta dan field-field), fungsi-fungsi (method, property, event, indexer, operator, constructor, destructor, dan static constructor) dan class dalam class.” ECMA p 211.

Beberapa buku OOP terkadang juga memberi istilah object sama dengan definisi Class ECMA, “object adalah kumpulan data dan fungsi” kira-kira demikian. Pada kenyataannya antara Class dan object itu berbeda. Saya lebih suka mendefinisikan object sebagai kumpulan kemampuan. Punya kemampuan apa. Sedangkan Class saya lebih suka menyebutnya sebagai blue-print dari object. Jadi design rumah saya adalah Class, sedang rumah saya adalah object yang dibuat berdasarkan design. Saya tidak peduli bagaimana rumah saya dibuat, yang penting buat saya adalah rumah ini memberi saya perlindungan, menjaga privasi saya, dan orang asing hanya akan sampai di ruang tamu. Ketika matahari sedang terik, rumah bisa melindungi saya dari panasnya matahari, begitupun saat badai dan hujan. Kalau saya ingin rumah saya melindungi saya dari gempa, rumah harus ditambah feature anti gempa. Kelak mungkin rumah kita perlu juga anti peluru dan anti bom. Saya tidak peduli bagaimana anti-anti tersebut dibuat. Yang penting buat saya adalah kumpulan fungsi-fungsi atau kemampuan rumah saya itu.

8.1 Deklarasi Class

Jadi Class lebih dekat kepada design dan object lebih dekat ke bentuk nyata. Lantas bagaimana mendesign Class? C# adalah bahasa anda dengan computer, dia bukan sulap. Selama anda mengikuti aturan bahasanya, ia akan menuruti anda, sekali anda bikin kesalahan bahasa ia akan memaki-maki anda. Bahkan terkadang mengumpat. ECMA memberikan aturan sebagai berikut:

Deklarasi Class :

```
attributesopt class-modifiersopt class identifier class-baseopt
class-body ;opt
```

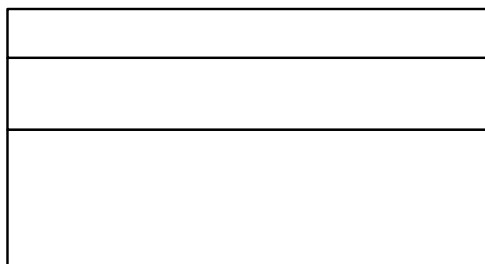
Bagian yang diakhiri dengan *opt* menunjukkan bagian ini adalah opsional, anda boleh memakainya dan boleh tidak. Ada tiga bagian yang harus ada, *class*, identifier dan *class-body*. *class* adalah keyword, jadi anda cukup menuliskannya, semuanya pake huruf kecil, "class". Identifier adalah nama dari Class yang akan kita buat, konvensi penamaan yang dipakai adalah pascal casing dan camel casing, saya tidak akan membahasnya disini, nama adalah bebas terserah anda, namun ada beberapa hal yang harus anda perhatikan, ini berkaitan dengan memory otak anda dan juga pembaca program anda/ atau teman anda. Pembaca program? Sekali anda membuat program bukan langsung sekali jadi, mungkin 1~2 tahun lagi anda harus melihat kembali, inipun jika anda benar-benar jenius. Bagaimana jika anda meninggal? Kawan anda atau teman-anda akan memaki-maki anda jika anda menamai Class anda secara serampangan. Nama yang bagus bagaimana? Gunakan nama benda yang ada didunia, tentunya yang terlibat dalam system/domain anda. *Class-body*, ini adalah tempat anda harus mendefinisikan kemampuan-kemampuan dari object yang dibangun berdasarkan Class ini. Anda adalah arsitek dan computer adalah pekerja bangunan anda.

Maaf kan saya kalau sampai paragraph ini anda menjadi sangat ngantuk dan pingin tidur. Saya juga begitu. Apalagi diseminan he..he..Baru pada bagian contoh biasanya saya terbangun.

8.2. Contoh Implementasi Class

Berikut kita akan coba membuat Class converter suhu. Harapannya, object yang dibuat dari Class ini akan mempunyai kemampuan antara lain:

- Mengubah suhu dari Celcius ke Fahrenheit dan sebaliknya.
- Mengubah suhu dari Celcius ke Kelvin dan sebaliknya.
- Mengubah suhu dari Fahrenheit ke Kelvin dan sebaliknya.



Gambar 8-1. Diagram class termometer.

Gambar 1 diatas adalah model dari Class termometer yang akan kita buat. Kotak diatas dibagi menjadi 3. Bagian atas menunjukkan nama Class, bagian ke dua saya mengisinya dengan *field/property*, tanda negative menunjukkan field tersebut hanya berlaku dalam class, atau bersifat pribadi. Bagian ke tiga saya isi dengan daftar fungsi object yang akan dibuat dari class ini. Tanda + menunjukkan fungsi-fungsi tersebut bisa diakses oleh object lain atau bersifat public.

_value dan _type masing-masing adalah data. _value menyimpan nilai angka, yaitu besarnya suhu. _type menyimpan nilai string yaitu F (Fahrenheit) dan C(Celsius). Nilai suhu selalu berpasangan dengan typenya F atau C. Misal 100 C, 212 F.

OOP mensyaratkan *information hiding* terhadap data, maka _type dan _value, kita buat hanya berlaku di dalam Class atau private. Untuk mengubah atau mengambilnya kita harus membuat fungsi/method. SetType adalah fungsi untuk mengubah nilai _type. SetValue adalah fungsi untuk mengubah nilai _value. Disini kita tidak membuat fungsi untuk mengambil. ToCels() adalah fungsi untuk mengubah nilai _value menjadi Celsius. ToFahr() adalah fungsi untuk mengubah nilai _value menjadi Fahrenheit.

Kedengarannya aneh dan rumit ya? Mungkin anda bertanya kenapa tidak dibuat menjadi tanda + semua jadi kita tidak perlu bikin fungsi pengubah dan pengambil. Nanti akan saya berikan contoh kenapa mesti demikian. Berikut ini adalah code dari model kita,

```

public class Termometer
{
    public Termometer()
    {
    }

    public void SetType(string type)
    {
        this._type=type;
    }
    public void SetValue(double val)
    {
        this._value=val;
    }
    public double ToCels()
    {
        if(this._type == "C")
        {
            return this._value;
        }
        else if(this._type=="F")
        {
            return 5*(this._value-32)/9;
        }
        else
        {
            return 0;
        }
    }
    public double ToFahr()
    {
        if(this._type == "C")
        {
            return 9*this._value/5 + 32;
        }
        else if(this._type=="F")
        {
            return this._value;
        }
        else
        {
            return 0;
        }
    }
}
private string _type;
private double _value;

```

} Class-body

```
}

```

Class yang kita buat ini kita beri nama termometer, public menunjukkan bahwa Class ini berlaku umum, artinya object manapun bisa memakainya.

Ini adalah bagian *constructor*, apabila kita tidak mendefinisikan *constructor*, Class akan memakai *constructor* default. *Constructor* inilah yang akan kita pakai untuk membuat object. Nama *constructor* harus sama dengan nama Class. *Constructor* tidak memiliki kembalian.

SetType adalah fungsi untuk memberi tanda object termometer apa yang dibuat, celsius atau fahrenheit. *void*, menunjukkan bahwa fungsi ini tidak ada nilai kembalian.

SetValue adalah fungsi untuk memberi nilai suhu pada object termometer.

ToCels adalah fungsi untuk mengubah nilai suhu menjadi nilai suhu lain dalam termometer celcius. Double, menunjukkan nilai keluaran dari fungsi ini adalah double. Kelak anda akan tahu keluaran ini betul-betul lepas dari apa yang kita inginkan, sebab typenya adalah double.

Sama dengan nomor 5, bedanya fungsi ini merubah suhu ke fahrenheit.

Ini adalah bagian data dari Class Termometer. Private, menunjukkan bahwa data tidak bisa diakses dari luar Class. Bagian data boleh dimana saja. Disini saya menaruhnya dibawah, dengan pertimbangan untuk monjolan fungsi dari objek.

Bagian berikut adalah contoh penggunaan Class termometer,

```
using System;
class UnitTest
{
    static void Main()
    {
        Termometer cels=new Termometer(); 1
        cels.SetType("C"); 2
        cels.SetValue(100); 3
        print(cels); 4
        Console.ReadLine();
    }
    private static void print(Termometer temp)
    {
        Console.WriteLine("Fahr : " + 5
                           temp.ToFahr().ToString()); 6
        Console.WriteLine("Cels : " +
                           temp.ToCels().ToString());
    }
}
```

Pertama-tama kita deklarasikan object cels dan langsung kita buat objeknya dengan keyword new kemudian kita panggil *constructor*-nya. Objek dibuat dengan pertama-tama memory dialokasikan untuk object ini kemudian *constructor* dipanggil.

Setelah objek termometer dibuat kemudian tipe termometer kita set menjadi C, yang berarti termometer kita ini adalah celcius.

Setelah type termometernya kita tetapkan, lantas kita set nilainya. Disini nilainya kita set menjadi 100. Yang berarti 100 C.

Kita panggil method print. Object termometer kita kirim ke bagian printing.

ToFahr kita panggil.
ToCels kita panggil.

8.3 Class untuk Object yang Tidak Berubah

Termometer kita susah ya cara pakainya?. Mesti new kemudian set ini set itu, yang berarti termometer kita bisa berubah-ubah kayak bungklon. Bisa tidak kita sekali buat, sekaligus diset apa type dan nilainya, dan tidak bisa diubah? Ok, mari kita ubah Class kita menjadi berikut ini:

```
public class Termometer
{
    public Termometer(string type, double val)
    {
        this._value =val;
        this._type=type;
    }
    public double ToCels()
    {
        if(this._type == "C")
        {
            return this._value;
        }
        else if(this._type=="F")
        {
            return 5*(this._value-32)/9;
        }
        else
        {
            return 0;
        }
    }
    public double ToFahr()
    {
        if(this._type == "C")
        {
            return 9*this._value/5 + 32;
        }
        else if(this._type=="F")
        {
            return this._value;
        }
        else
        {
            return 0;
        }
    }
    private string _type;
    private double _value;
}
```

Di listing diatas tanpa parameter saya hapus, demikian juga dengan setValue dan setType. Kemudian saya buat *constructor* baru tetapi dengan dua parameter yaitu type dan value. Lihat 1. Dengan cara seperti ini, data Class hanya bisa diset saat konstruksi,

setelah itu tidak ada aksi pengubah yang diizinkan. Akibatnya Class seperti ini statenya selalu tetap, *immutable*.

Bagian berikut adalah contoh penggunaan Class termometer yang telah kita ubah,

```
using System;
class UnitTest
{
    static void Main()
    {
        Termometer cels=new Termometer("C", 100) 1
        print(cels);

        Console.ReadLine();
    }
    ....
}
```

8.4. Class Abstrak

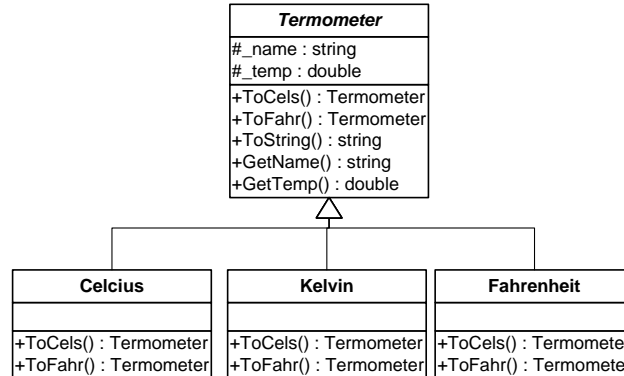
Salah satu tanda kita telah menggunakan OOP dengan benar adalah seberapa banyak kita meninggalkan statement if. Coba kita tengok lagi Class termometer yang telah kita buat.

1. Tidak ada kaitan antara nama dengan besaran suhu. 2. Mestinya secara intuitif ketika kita mengubah ke Celsius yang didapat adalah Celsius bukan double, sehingga ketika nilai keluaran ini kita tangkap, yang kita dapat adalah sebuah nilai bebas. 3. Kita masih ada 2 termometer lagi Kelvin dan Rheamour, dengan cara diatas bukan pekerjaan yang mudah untuk menambahkan dua termometer ini.

Termometer. Apa yang terbayang di benak anda ketika mendengar kata ini. Alat pengukur suhu. Disini kita juga menanbahkan kemampuan husus sebagai alat pengkonversi suhu. Tetapi ketika anda hendak mengukur suhu yang anda butuhkan adalah wujud nyata dari termometer misalnya, Celsius, Fahrenheit, Kelvin. Disini termometer adalah abstrak. Sedangkan Celsius dan kawan-kawan adalah kongkrit. Secara abstrak termometer mempunyai kemampuan mengukur dan mengkonversi. Anda tidak akan bisa mengukur atau mengkonversi dengan tool yang hanya dalam teori atau masih dalam fikiran dan kata-kata.

Anda harus mencari tool yang nyata, kongkrit. Tool tersebut dikatakan termometer jika dia mempunyai kemampuan yang sama dengan termometer dalam fikira/teori. Dengan demikian Celsius harus mengimlementasikan, menyediakan kemampuan-kemampuan termometer abstrak jika ingin disebut termometer. Demikian juga dengan Kelvin, Fahrenheit dll. Dengan cara seperti ini pengguna termometer tidak akan kebingungan, sebab semua thermometer mempunyai cara/kemampuan minimal yang sama. Mengapa saya katakana minimal, sebab ada kemungkinan pabrik pembuat tool akan menambahkan kemampuan lain yang tidak ada dalam terori termometer.

Menghubungkan antara yang abstrak dengan yang kongkrit dalam OOP disebut sebagai menggeneralisasi, hubungannya disebut *inheritance*, kadang saya menyebutnya turunan, dengan pengertian tidak 100%. Model hubungannya terlihat dalam diagram berikut ini:



Gambar 8-1. Hubungan inheritance diimplementasikan dalam bentuk anak panah, tanpa ekor

Hubungan Class termomete kita wujudkan dalam code sebagai berikut.

```

namespace EBook. OpenSource. Bab8
{
    public abstract class Termometer
    {
        public Termometer(double temperatur)
        {
            this._temp =temperat ❶;
        }
        public abstract Termometer ❷ ToCels();
        public abstract Termometer ❸ ToFahr();
        public override string ToString()
        {
            return this.GetTemperatur(). ToString() + ❹
                "derajat " + this.GetName();
        }
        public string GetName()
        {
            return this._name; ❺;
        }
        public double GetTemperatur()
        {
            return this._temp;
        }
        protected double ❻ temp;
        protected string name;
    }

    public class Cel s: Termometer
    {
        public Cel s(double Temperatur): base(Temperatur)
    }
}
    
```

```

        {
            base._name="Cel s i s";
        }
        public override Termometer ToCel s()
        {
            return this;
        }
        public override Termometer ToFahr()
        {
            double fahrTemp;
            fahrTemp=9*this.GetTemperatur()/5 + 32;
            Fahr f=new Fahr(fahrTemp);
            return f;
        }
    }
    public class Fahr: Termometer
    {
        public Fahr(double Temperatur): base(Temperatur)
        {
            base._name="Fahrenhei t";
        }
        public override Termometer ToCel s()
        {
            double cel sTemp;
            cel sTemp=5*(this.GetTemperatur()-32)/9;
            Cel s c=new Cel s(cel sTemp);
            return c;
        }
        public override Termometer ToFahr()
        {
            return this;
        }
    }
    public class Kel v: Termometer
    {
        public Kel v(double Temperatur): base(Temperatur)
        {
            this._name="Kel vi n";
        }
        public override Termometer ToCel s()
        {
            double cel sTemp;
            cel sTemp=this.GetTemperatur() - 273.16;
            Cel s c=new Cel s(cel sTemp);
            return c;
        }
        public override Termometer ToFahr()
        {
            return this.ToCel s().ToFahr();
        }
    }
}

```

Konstruktor Class abstrak berparameter tunggal, temperatur. Class ini akan kita buat menjadi *base-class* dari Class-Class kongkrit. Class ini menyimpan data tipe termometer dan suhu termometer. Kedua data ini *modifier*-nya **protected** yang berarti bisa diakses turunannya, tetapi tidak dari Class yang bukan turunannya.

Definisi method *abstract* dari ToCels. Method *abstract* tidak ada implementasi, hanya definisi saja. Class kongkrit yang diturunkan dari Class ini harus mengimplemenasikan fungsi yang masih abstrak ini.

Defini method *abstract* dari ToFahr.

Kita meng-*override* method ToString dari class object. Di C# semua Class sebetulnya diturunkan dari Class object ini. Lebih jauh tentang override kita akan bahas dibagian lain dari bab ini.

Ini adalah aksesor untuk nama. Anda bisa juga menggunakan properti, tapi saya tidak akan membahas dalam bab ini.

Data kita deklarasikan *protected* agar bisa diakses dari Class turunannya.

Perhatikan bagaimana Class kongkrit diturunkan dari Class abstrak.

Nama_Class : nama_Class_bastrak.

Perhatikan *_name* yang berada di Class abstrak bisa diakses dari turunannya.

Sementara suhu kita bisa set dari base *constructor*-nya.

Anda juga bisa perlakukan suhu seperti *_name*.

Dua method abstrak yang harus diimplementasikan adalah ToCels dan ToFahr. Untuk mengimplementasikannya kita harus memakai keyword **override**. Perhatikan bagaimana method abstrak diimplementasikan.

Impelementasi method ToFahr. Perhatikan keluaran dari method ini *Termometer*, padahal kita meng-*create* objek Fahrenheit. Hal ini bisa dilakukan karena class Fahrenheit diturunkan dari Class abstrak termometer.

```
using System;
using EBook.OpenSource.Bab8;
namespace ConTest
{
    class UnitTest
    {
        static void Main()
        {
            TestTemp1();
            Console.ReadLine();
        }

        private static void TestTemp1()
        {
            Cels c=new Cels(100);
            Fahr f=new Fahr(212);
            Console.WriteLine("====Cels====");
            printTemp1(c);
            Console.WriteLine("====Fahr====");
            printTemp1(f);
            Console.WriteLine("====Kelv====");
            Kelv k=new Kelv(373.16);
            printTemp1(k);
        }

        private static void printTemp1(Termometer temp1)
        {
            Console.WriteLine("{0} sama dengan {1}",
                temp1.ToString(),
                temp1.ToCels().ToString());
            Console.WriteLine("{0} sama dengan {1}",
                temp1.ToString(),
                temp1.ToFahr().ToString());
        }
    }
}
```

```
    }  
  }  
}
```

Perhatikan objek celsius dan fahrenheit kita buat langsung dengan memakai Class kongkrit.

Perhatikan bagaimana method `printTemp1` me-refer kedua object hanya melalui Class abstraknya. Konsep ini hampir sama dengan konsep *interface* yang akan kita bahas di bab selanjutnya.

8.5 Inheritance

Hubungan antara yang abstrak, missal Celsius, dengan yang kongkrit kita sebut sebagai *inheritance* atau turunan. Semua yang `public` dan `protected` secara otomatis menjadi bagian dari turunannya. Apabila anda menghendaki perilaku/fungsi/kemampuan yang berbeda dari induknya anda harus melakukan *overriding*. Anda bisa melakukan *overriding* jika fungsi di atasnya (induknya) tidak di seal.

Semua fungsi/kemampuan yang berada didalam induknya yang ditandai dengan abstrak harus diimplementasikan dalam turunan semuanya tanpa kecuali. Sehingga dengan cara ini fasilitas/kemampuan minimal atau dasar dari induknya terpenuhi. Sehingga cukup disebut sebagai kongkrit dari yang abstrak.

Lebih jauh tentang inheritance, akan kita bahas secara terbisah dibab-bab selanjutnya.

Bagian diatas menunjukkan contoh bagaimana Class dideklarasikan dan kemudian di buat objeknya, kemudian sedikit saya tunjukkan bagaimana OOP memudahkan kita membuat program dengan mengeliminasi statement `if`.

8.6 Lebih Jauh dengan Class

Bagian awal dari bab ini saya telah menunjukkan bagaimana Class dideklarasikan. Bila *attribute* kita singkirkan terlebih dahulu, saya harapkan anda mempelajarinya kelak, maka deklarasinya akan seperti ini:

```
Class-modifier class identifier : class-base  
{  
  //Class-members;  
}
```

Class-modifier: `new`, `public`, `protected`, `internal`, `private`, `abstract`, `sealed`.

`New` tidak akan saya jelaskan disini. `Public`, `protected`, `internal`, `private` mengontrol akses ke Class. `Internal` membatasi akses hanya dalam satu assembly, assembly lain tidak bisa mengakses Class ini. *Abstract* dan *Sealed* saya akan terangkan setelah ini. Modifier boleh ditulis berdua misal, `public abstract` seperti contoh class thermometer diatas.

8.6.1 Class Abstrak

Class abstrak digunakan untuk menunjukkan bahwa Class ini belum selesai, masih abstrak belum kongkrit. Class ini biasanya digunakan sebagai *base class*. Seperti contoh termometer diatas, Class abstrak tidak bisa di-*create* menjadi objek, tetapi bisa me-*refer* objek kongkrit yang Classnya diturunkan dari dirinya. Class abstrak boleh memiliki *abstract member*, seperti contoh diatas :ToCels() dan ToFahr(). Class abstrak **tidak** boleh di seal. Seal akan kita bahas bagian berikut ini.

Class kongkrit yang diturunkan dari Class abstrak harus mengimplementasikan semua *abstract member*. Class Celsius dan Fahrenheit harus mengimplementasikan ToCels dan ToFahr dengan cara meng-*override*.

Contoh .:

```
abstract class A
{
    public abstract void F();
}

abstract class B:A
{
    public void G()
    {
    }
}

class C: B
{
    public override void F() {
        //Implementasi sesungguhnya
    }
}
```

Ini adalah contoh dari ECMA, p 212. A adalah Class abstrak dengan *abstrak member* F(). *Abstract member* tidak ada aktifitas apapun, alias kosongan. Class B diturunkan dan Class A, tetapi karena B adalah Class abstrak maka ia tidak perlu mengimplemantasikan F() ini tidak berarti B tidak boleh mengimplemenasikan F(). Tetapi jika B mengimplemantasikan F() maka C tidak wajib mengimplementasikan F(). Class B menambah method G(). Karena Class C diturunkan dari B sedang B diturunkan dari A maka C harus mengimplemantasikan F(). Lebih jelas anda bisa lihat contoh termometer diatas.

8.6.2. Sealed Class

Sealed class adalah Class yang memakai modifier **sealed**. Class yang di-*seal* tidak bisa diturunkan atau tidak bisa menjadi *base class* dari Class yang lain.

Contoh:

```
sealed class C: B
{
    public override void F()
    {
    }
}
```

```
        Console.WriteLine("FROM C.F()");
    }
}
class D: C
{
}
```

Class D tidak bisa di kompil karena C adalah *sealed class*.

8.6.3. Class Member

Class member dibagi menjadi beberapa kategori:

- Konstanta
- Field
- Method
- Properties
- Event
- Indexer
- Operator
- Constructor
- Destructor

Disini kita tidak akan membahas properties, event, indexer dan operator karena buku ini ditujukan untuk para pemula.

8.6.3.1. Konstanta

Konstanta adalah anggota Class yang bernilai tetap. Nilai konstanta diperhitungkan pada saat kompilasi program. Konstanta dideklarasikan sebagai berikut:

Attribute_{opt} modifiers_{opt} **const** type deklarasi

Attribute kita tidak akan bahas disini.

Modifiers: new, public, protected, internal, private.

Public : semua objek boleh akses tanpa batas.

Private: hanya berlaku dalam Class.

Protected: hanya boleh diakses Class sendiri dan turunannya.

Protected: hanya boleh diakses Class-Class dalam satu assembly.

Declarasi:

```
Identifer = ekspresi
Identifer=ekspresi , Identifer =ekspresi
```

Contoh

```
class
{
    public const double X=1.0, Y=2.0, Z=3.0;
}
```

Anda boleh juga menulisnya

```
class A
{
    public const double X=1.0;
    public const double Y=2.0;
    public const double Z=3.0
}
```

Ingat konstanta diperhitungkan saat kompilasi bukan saat run program, sehingga kita tidak bisa memberikan nilai yang tidak bisa diperhitungkan saat kompilasi. Konstanta boleh ada ketergantungan dengan konstanta Class lain dalam satu assembly maupun tidak sepanjang ketergantungannya tidak sirkular. Contoh berikut ini adalah legal:

```
class A
{
    public const int X=B.Z + 1;
    public const int Y=10;
}
class B
{
    public const int Z =A.Y + 1;
}
```

Saat kompilasi pertama-tama dilakukan evaluasi terhadap A.Y, kemudian B.Z baru kemudian A.X. Hasilnya adalah 10, 11, 12. Saya tidak menyarankan anda melakukan hal ini, ingat prinsip OOP, *information hiding*. Anda boleh nekat bila memang dalam kondisi terpaksa tapi masih dalam satu assembly.

Misalnya dalam Class termometer, kita ingin nilainya ada pada batas tertentu, karena termometer kita adalah termometer untuk mengukur suhu badan. Kita akan berikan batas antara 0 sampai 100 derajat celsius. Kemudian kita akan coba gagalkan inialisasi yang melebihi 100 atau kurang dari 0.

```
public class Cels: Termometer
{
    private const double _min=0;
    private const double _max=100;

    public Cels(double Temperatur): base(Temperatur) {
        if(! (Temperatur <=Cels._max && Temperatur>=0))
            throw new ApplicationException("Overflow Temperatur.");
        base._name="Cel si us";
    }
    .....
}
```

Kita tidak bisa mengambil nilai konstanta dengan keyword **this**. Seperti terlihat pada contoh diatas anda harus memanggil nama Classnya jadi harus Cels._max dan Cels._min.

Terkadang kita ingin membuat konstanta dari sebuah Class yang mesti diinstan. Hal ini tidak bisa dilakukan mengingat, konstanta dievaluasi saat kompilasi. Untuk melakukan hal ini kita harus memakai field.

8.6.3.2. Field

Field adalah anggota Class yang merupakan variable. Dalam contoh class temperatur, `_temp` dan `_name` kita sebut sebagai field. Field dideklarasikan sebagai berikut:

```
Attributesopt modifieropt type decl arasi
```

Modifier: new, public, protected, internal, private, static, readonly, volatile

Declarasi :

Identifier;

Identifier=inisialisasi variabale;

Identifier=Inisialisasi variabale, identifier = inisaiaisasi variable;

Dalam OOP field hanya saya pakai untuk menyimpan data. Sedangkan data ini sifatnya disembunyikan. Untuk mencapai data tersebut gunakan *accessor* atau method untuk menanganinya. Kelihatannya ini terlalu kompleks, tetapi ketika program anda menjadi besar, konsep ini akan sangat banyak membantu.

Terkadang kita ingin membuat suatu konstanta dari instance suatu object, tetapi karena konstanta dieksekusi saat kompilasi hal ini tidak dapat dilakukan. Misal, kita ingin sebuah Class berisi konstanta warna hitam, putih, merah, hijau, dan biru, tetapi karena warna-warna ini dibuat dari Class yang objeknya mesti diinstan, akibatnya kita tidak bisa menulisnya sebagai:

```
const Color Black=new Color(0,0,0);
```

Oleh karena itu kita akan gunakan static field yang *readonly*. Perbedaan antara static dan instance field adalah berapa kalipun anda menginstan Class menjadi object, static field akan tetap satu, sedangkan instan field akan sebanyak objek dari Class tersebut. Maka dari itu orang kemudian lebih suka menyebut static field sebagai milik Class dari pada object. Berikut ini adalah contohnya,

```
public class Color
{
    public static readonly Color Black=new Color(0,0,0);
    public static readonly Color White=new Color(255,255,255);
    public static readonly Color Red=new Color(255,0,0);
    public static readonly Color Green=new Color(0,255,0);
    public static readonly Color Blue=new Color(0,0,255);

    private byte red, green, blue;
    public Color(byte r, byte g, byte b) {
        red=r;
        green=g;
        blue=b;
    }
}
```

Berapapun anda membuat objek Color, instan dari Black, White, Red, Green dan Blue tetap satu. Karena warna-warna ini bukan milik objek anda tidak bisa me-*refer* dengan cara `object.Black` tetapi anda harus me-*refer*-nya sebagai `Color.Black`. Nanti and akan banyak sekali berhadapan dengan masalah ini ketika anda menginginkan dalam memory hanya boleh ada satu instan, dalam hal ini anda harus melihat pola *Singleton Pattern*.

Kembali ke field, apabila kita tidak melakukan inisialisasi terhadap field, maka nilai inisialnya adalah default dari type tersebut.

Contoh:

```
using System;
class Test
{
    static bool b;
    int i;
    static void Main()
    {
        Test t=new Test();
        Console.WriteLine("b= {0}, i= {1}", b, t.i);
    }
}
```

Hasilnya adalah b=False, i=0.

8.6.3.3 Method

Method adalah anggota Class yang berisi baris perintah perhitungan atau aksi yang bisa dipanggil oleh Class atau objek. Method dideklarasikan sebagai berikut:

```
Attributeopt modifieropt return-type name(parameteropt)
{
    . . . . Statements;
}
```

Modifier: new, public, protected, internal, private, static, virtual, sealed, override, abstract, extern.

Return-type: type, void. Void digunakan jika method tidak ada nilai kembalian.

Parameter Modifier: ref, out, params.

Dari contoh termometer kita bisa ambil contoh method:

```
class Celsius: Termometer
{
    . . .
    public Termometer ToFahr()
    {
        double fahrTemp;
        fahrTemp=9*this.GetTemperatur()/5 + 32;
        Fahr f=new Fahr(fahrTemp);
        return f;
    }
    . . . . .
}
```

Disini kita tidak memakai attribute, ini saya tinggalkan untuk anda..he..he...Public berarti *no limited access*, akses tanpa batas, siapapun kapanpun. Method ini mempunyai keluaran berupa objek bertipe Termometer, karena itu dibagian akhir aktifitasnya saya kembalikan dengan kata-kata **return**. Jika method anda tidak ada nilai kembalian cukup katakan pada compilernya **void**. Misal,

```
public void ToFahr()
{
    //... anda tidak perlu kasih return value disini.
```

```
}
```

Selanjutnya kita akan membahas method yang berparameter. Parameter adalah nilai yang kita lewatkan pada saat kita memanggil method. Misalnya kita mempunyai fungsi Sin, Cos, Tan. Sin(30). Nilai 30 inilah yang saya sebut sebagai parameter. Terkadang method membutuhkan lebih dari satu parameter. Terkadang method yang sama mempunyai parameter yang tidak tentu jumlahnya. Terkadang kita juga ingin keluaran method lebih dari satu, sehingga kita tidak cukup menaruhnya dibagian return value (type). Di C# ada 4 macam parameter:

- Value Parameter.
- Reference Parameter.
- Output Parameter.
- Parameter Array.

Value Parameter

Parameter yang dideklarasikan tanpa parameter modifier disebut sebagai value parameter. Kita harus hati-hati disini, di C# kita ada dua jenis type, **Value Type** dan **Reference Type**. Value Type : struct type (tipe numerik dan tipe bool), Reference Type: class, interface, delegate. Apabila yang kita lewatkan adalah value type maka nilainya akan ditransfer ke local variable, perubahan terhadap nilai ini tidak akan merubah state dari variabel aslinya, ini kebalikan dari reference parameter. Namun bila yang kita lewatkan adalah Reference Type maka yang terlewatkan adalah reference dari type, akibatnya bila kita ubah state dari object ini maka state aslinya juga akan berubah. Agar lebih jelas saya akan berikan contoh:

```
using System;
class A
{
    private int a;

    public int Ambil ()
    {
        return this.a;
    }
    internal void Ubah(int obj Baru)
    {
        this.a=obj Baru;
        Console.WriteLine("A di ubah jadi : " + this.a);
    }
}

class Uni tTest
{
    static void UbahA(A a)
    {
        a.Ubah(10);
    }
    static void Nai kkanB(int b)
    {
        b=b+1;
        Console.WriteLine("B di tambah 1 ");
    }
    static void Mai n()
    {
        A a=new A();
        int B=0;
```



```

        Console.WriteLine("A sama dengan 0 : " +
            a.Ambil().Equals(0).ToString());
        UbahA(a);
        Console.WriteLine("A sama dengan 10 : " +
            a.Ambil().Equals(10).ToString());
        Console.WriteLine("B sama dengan 0 : " +
            B.Equals(0).ToString());
        NaikkanB(B);
        Console.WriteLine("B sama dengan 1 : " +
            B.Equals(1).ToString());
        Console.Read();
    }
}

```

Outputnya akan seperti ini:

```

A sama dengan 0 : True
A di ubah jadi : 10
A sama dengan 10 : True
B sama dengan 0 : True
B di tambah 1
B sama dengan 1 : False

```

Saat diinstan objek a akan memiliki nilai a=0, default value. Demikian juga dengan objek B. Ketika object a diubah nilainya oleh UbahA(), nilai a diobjek a berubah menjadi 10. Tetapi nilai B tidak berubah sama sekali ketika method Naikkan() dipanggil. Hal ini dikarenakan objek a yang kita lewatkan ke UbahA() adalah *reference type*. Sedangkan nilai yang kita lewatkan ke NaikkanB() adalah *value type*. Agar B berperilaku sama dengan *reference type*, maka kita harus memakai *reference parameter*.

Reference Parameter

Reference parameter adalah parameter yang dideklarasikan dengan modifier **ref**. Sebelum variable dilemparkan ke parameter variable tersebut harus diinisialisai. Contoh:

```

using System;
class Uni tTest
{
    static void NaikkanB(1 ref int b)
    {
        b=b+1;
        Console.WriteLine("B di tambah 1 ");
    }
    static void Mai n()
    {
        2 int B=0;
        Console.WriteLine("B sama dengan 0 : " +
            B.Equals(0).ToString());
        NaikkanB(3 ref B);
        Console.WriteLine("B sama dengan 1 : " +
            B.Equals(1).ToString());
        Console.Read();
    }
}

```

Hasil nya:
 B sama dengan 0 : True

B di tambah 1
 B sama dengan 1 : 4ue

Perhatikan deklarasi method berikut ini, agar menjadi *reference parameter* kita cukup menambah **ref** didepan type.

```
static void NaikkanB(ref int b)
```

Variable B kita *create* dan sekaligus kita inialisasi.

Kemudian dimethod pemanggilnya juga harus kita tambahkan keyword **ref** sebelum variable yang kita kirim.

Akibatnya B ikut berubah dari nol menjadi 1. Coba anda bandingkan dengan contoh sebelum ini.

Output parameter

Parameter yang diawali dengan keyword **out** kita sebut sebagai *output parameter*. *Output parameter* biasa digunakan jika kita ingin method kita memiliki lebih dari satu keluaran. Berbeda dari ref, variabel yang dilempar ke *output parameter* TIDAK perlu diinisialisai terlebih dahulu. Contoh:

```
using System;
class PemisahPath
{
    public void Pisah(string path, out string 1 r, out string
name)
    {
        int i=path.Length;
        while(i>0)
        {
            char ch = path[i-1];
            if(ch=='\' || ch=='/' || ch==':') break;
            i--;
        }
        dir =path.Substring(0,i);
        name=path.Substring(i);
    }
}

class UnitTest
{
    static void Main()
    {
        string dir, name;
        PemisahPath p=new PemisahPath();
        p.Pisah("C:\\OPS\\Fatur\\Job.txt", out dir 3 out name);
        Console.WriteLine(dir);
        Console.WriteLine("Direktornya C:\\OPS\\Fatur\\ : {0}",
            dir=="C:\\OPS\\Fatur\\"? "Betul" : "Salah");
        Console.WriteLine("Nama File Job.txt : {0}",
            name=="Job.txt"? "Betul" : "Salah");
    }
}
```

Hasilnya:
 > Executing: C:\Program Files\ConTEXT\ConExec.exe
 "D:\fatur\IS.Camp\ParamOUT\ParamOUT.exe"

C:\OPS\Fatur\

```
Di rektornya C:\OPS\Fatur\ : Betul
Nama File Job.txt : Betul
> Execution finished.
```

Pertama kita bikin class dengan method Pisah. Perhatikan bagaimana out digunakan. Kedua kita deklarasikan variabelnya. Kita tidak perlu menginisialisasi, yang saya maksud memberi nilai terlebih dahulu. Perhatikan bagaimana out dipakai saat pemanggilan.

Parameter Array

Terkadang kita ingin mengirimkan parameter berupa array yang berarti tipenya sama tetapi jumlah nilainya beragam. Nah kita harus mendeklarasikan parameter ini dengan awalan **params**. Param tidak bisa digabung dengan out ataupun ref. *Parameter array* harus dideklarasikan pada posisi paling belakang jika ada banyak parameter. Selain itu arraynya harus berdimensi satu, jadi `int[,]` tidak boleh. Contoh:

```
using System;
class ParamArray
{
    public void SetArray(int test, params int[] args)
    {
        this.test=test;
        this.arrayParam=args;
    }
    public int Count()
    {
        return arrayParam.Length;
    }
    private int test;
    private int[] arrayParam;
}

class Uni tTest
{
    static void Main()
    {
        ParamArray paramArr=new ParamArray();
        int[] arr={7, 8, 9, 10};
        paramArr.SetArray(5, arr);
        Console.WriteLine("Jumlah elemen dalam array 4, {0}",
            paramArr.Count().Equals(4));
        paramArr.SetArray(5, 1, 2, 3, 4, 5);
        Console.WriteLine("Jumlah elemen dalam array 5, {0}",
            paramArr.Count().Equals(5));
        paramArr.SetArray(5);
        Console.WriteLine("Jumlah elemen dalam array 0, {0}",
            paramArr.Count().Equals(0));
    }
}
```

```
Hasil nya:
> Executing: C:\Program Files\ConTEXT\ConExec.exe
"D:\fatur\I S. Camp\ParamArray\ParamArray.exe"
```

```
Jumlah elemen dalam array 4, True
```

```
Jumlah elemen dalam array 5, True  
Jumlah elemen dalam array 0, True  
> Execution finished.
```

Perhatikan baik-baik bagaimana saya mendeklarasikan **params**.

Ada tiga test yang saya buat, bagian ini saya buat deklarasi sekaligus inialisasi array. Kemudian saya lempar masuk ke Method.

Bagian ini agak aneh, tetapi inilah mungkin alasan mengapa param array mesti dideklarasikan di bagian paling bontot dan harus array tunggal.

Bagian ini saya test seandainya saya tidak memberikan array. Oh..Tuhan, dia ngerti juga.

Virtual Method

Apakah itu Virtual Method? Saya susah menceritakannya. Lebih baik saya akan kasih contoh, supaya anda ikut merasakan rasa seperti saya, ceile...Ok. Perhatikan contoh berikut:

```
using System;  
class VirtualBase  
{  
    public int NonVirtual()  
    {  
        return this.1NonVirtual;  
    }  
    public virtual int Virtual()  
    {  
        return this.2Virtual;  
    }  
    private int _nonVirtual=1;  
    private int _virtual=2;  
}  
  
class TurunanVirtual:VirtualBase  
{  
    new public int NonVirtual()3  
    {  
        return this._nonVirtual;  
    }  
    public override int Virtual()4  
    {  
        return this._virtual;  
    }  
    private int _nonVirtual=3;  
    private int _virtual=4;  
}  
  
class UnitTest  
{  
    static void Main()  
    {  
        TurunanVirtual tv=new TurunanVirtual();  
        VirtualBase vb=tv;5  
  
        Console.WriteLine("Virtual Base NonVirtual ()=1, {0}",  
            vb.NonVirtual().Equals(1));  
        Console.WriteLine("Virtual Base Virtual ()=4, {0}",  
            vb.Virtual().Equals(4));  
        Console.WriteLine("Turunan Virtual NonVirtual ()=3, {0}",
```

```

        tv.NonVirtual().Equals(3));
        Console.WriteLine("Turunan Virtual Virtual ()=4, {0}",
            tv.Virtual().Equals(4));
    }
}

```

Hasilnya:

```

> Executing: C:\Program Files\ConTEXT\ConExec.exe
"D:\fatur\I.S. Camp\Virtual Method\Virtual Method.exe"

```

```

Virtual Base NonVirtual()=1, True
Virtual Base Virtual()=6, True
Turunan Virtual NonVirtual()=3, True
Turunan Virtual Virtual()=4, True
> Execution finished.

```

Pertama kita buat non virtual method (tidak ada keyword virtual) di *base class*.

Selain itu kemudian kita juga membuat virtual method (perhatikan keyword virtual) di *base class*.

Kemudian kita buat turunan dari *base class* dan kita buat definisi ulang dari method non virtual, perhatikan keyword new. Non virtual method tidak bisa kita override.

Kemudian kita override method virtual dari *base class* dengan definisi baru. Perhatikan kata override disitu.

Kita *create* objek turunan dari, setelah itu kita buat *base class* me-*refer* ke objek turunan. Perhatikan kita tidak membuat instan objek base kita hanya me-*refer*.

Perhatikan walaupun kita panggil method dari *base class* yang terpanggil tetap anaknya, inilah gunanya override. Berbeda kan dengan yang non virtual? Jadi ketika kita membuat method virtual, kompiler akan mencari anak yang mengoverride method tersebut, kemudian method yang di anak itulah yang diambil. CATATAN, ketika kita membuat Class sebetulnya kela kita diturunkan dari Class object. Dari Class object ini ada beberapa method yang suatu ketika perlu anda override, Equals() kalau anda mengoverride method ini anda harus juga mengoverride method GetHashCode(). Selain itu anda juga bisa mengoverride ToString().

Bagaimana memanggil fungsi asli? Gunakan keyword **base.Method()**. Contoh:

```

class TurunanVirtual : Virtual Base
{
    ...
    public int Virtual Base()
    {
        return base.Virtual ();
    }
    ...
}

```

Override Method

Anda bisa lihat penjelasan dibagian Virtual Method. Yang perlu digarisbawahi bahwa hanya method virtual yang bisa dioverride. Agar method tidak bisa dioverload kita harus menambahkan modifier **sealed**, sehingga methodnya akan berubah menjadi sealed method. Contoh:

```
class TurunanVirtual : VirtualBase
{
    ...
    public sealed override 1 Virtual ()
    {
        return this._virtual;
    }
    .....
}
class TestSealed : TurunanVirtual
{
    public override 2 Virtual ()
    {
        return 5;
    }
}
```

Error Compilasi :

```
> Executing: C:\Program Files\ConTEXT\ConExec.exe
"C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322\csc.exe"
D:\fatur\IS.Camp\VirtualMethod\*.cs
```

```
Microsoft (R) Visual C# .NET Compiler version 7.10.6001.4
for Microsoft (R) .NET Framework version 1.1.4322
Copyright (C) Microsoft Corporation 2001-2002. All rights
reserved.
```

```
VirtualMethod.cs(34,27): error CS0239: 'TestSealed.Virtual()' :
cannot override inherited member 'TurunanVirtual.Virtual()'
because it is sealed
VirtualMethod.cs(21,34): (Location of symbol related to previous
error)
> Execution finished.
```

Perhatikan sealed method dideklarasikan.

Sealed method kita coba untuk dioverride.

Error yang menyatakan tidak bisa dioverride karena di seal.

Abstract Method

Abstract method adalah method yang didefinisikan dengan keyword **abstract**. *Abstract method* hanya boleh dideklarasikan di Class abstrak. Method abstrak tidak ada implementasi aktual. Semua class yang diturunkan dari Class abstrak ini harus mengoverride semua method abstrak yang ada. Perhatikan contoh di Termometer bagian ke dua.

Method Overloading

Terkadang kita menginginkan method dengan nama yang sama tapi dengan parameter yang berbeda-beda. Nah method ini kita sebut sebagai Method Overloading. Dalam C# hal ini diizinkan. Yang harus diperhatikan adalah type bukan nama. Contoh:

```
class MethodOverLoading
{
    public void Test(int a, string s)
    {
    }
    public void Test(string a, int s)
    {
    }
    public void Test(string s, int a, byte b)
    {
    }
    public void Test(byte b, string s)
    {
    }
}
```

1 dan 2 kalau menurut saya sama ini Cuma posisinya aja beda. Sama compiler dibedakan lho. Nama enggak ngaruh, yang ngaruh adalah type dan posisi. Posisi menentukan prestasi he..he... Coba yang nomor dua dibuat begini

```
public void Test(int s, string a)
{
}
}
```

Error tidak? Kalau tidak, pinjam dong *compiler*-nya.

Tujuannya apa kita melakukan overloading? Bukan untuk keren-kerenan lho. Dengan method yang sama anda bisa memanggilnya dengan berbagai cara. Suatu ketika anda buat tidak ada parameternya, tiba-tiba perlu parameter, paling tidak anda tidak perlu puyeng nyari dependensinya. Kenapa gak pake optional saja kayak di VB. He..he..ini soal style mas. Rambut saya kriting enggak mungkin dipanjangin entar jadi krebo. Kalau pake optional kita perlu check ada nilai atau pake nilai default, ini tidak menyenangkan dan coding kita enggak bersih. Dan perlu anda ketahui, antar method bisa saling panggil memanggil lho!! Enggak kebayang saya kalau pake Optional. Contoh:

```
using System;
class MethodOverLoading
{
    public void Test()
    {
        Console.WriteLine("Test");
        this.Test(12);
    }
    public void Test(int a)
    {
        Console.WriteLine("Test(int a)");
        this.Test(a, "Rumah ku");
    }
    public void Test(int a, string s)
    {
    }
}
```

```

        Console.WriteLine("Test(int a, string s) " + s);
        this.Test(s, a, 1);
    }
    public void Test(string s, int a, byte b)
    {
        Console.WriteLine("Test(string s, int a, byte b)");
        Console.WriteLine(s + " " + a + " " + b);
    }
}
class UnitTest
{
    static void Main()
    {
        MethodOverLoading mo =new MethodOverLoading();
        mo.Test();
    }
}

```

Hasilnya:

```

> Executing: C:\Program Files\ConTEXT\ConExec.exe
"D:\fatur\I S. Camp\OverLoadMethod\MethodOverLoading.exe"

```

```

Test
Test(int a)
Test(int a, string s) Nomor Rumah ku
Test(string s, int a, byte b)
Nomor Rumah ku 12 1
> Execution finished.

```

Bisa merasakan implikasi apa yang bisa ditimbulkan dari ini? Kalau tidak...mungkin anda lebih cerdas dari saya. I'm dummy. Cocoknya nulis C# for Dummies.

Constructor Overloading

Ini sama dengan method overloading bedanya ini dibuat terhadap konstruktor. Contoh: Dari contoh termometer

```

public class Termometer
{
    public Termometer(string type, double val)
    {
        this._value =val;
        this._type=type;
    }
    public Termometer()
    {
    }
    .....
}

```

Dengan demikian kita bisa menginstance objek dari class ini dengan 2 cara.

```

Termometer t1=Termometer();
Termometer t2=Termometer("C", 100);

```


9. Inheritance

Norman Sasono

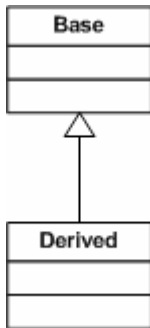
Pada bab ini akan dijelaskan tentang class inheritance dengan lebih detail setelah sebelumnya anda mendapat pengantar tentang inheritance di bab tentang OOP. Setelah membaca bab ini diharapkan anda lebih memahami bagaimana konsep inheritance dari OOP ini diterapkan secara sintaksis di C#. Termasuk beberapa aturan implementasinya.

9.1 Class dan Inheritance

Inheritance, di dalam OOP dijelaskan sebagai kemampuan sebuah object untuk meng-inherit atau mewarisi data dan functionality dari parent class-nya. Dengan inheritance, anda dapat membuat sebuah class baru dari class-class yang sudah ada daripada membuat sendiri dari awal sebuah class yang benar-benar baru. Parent class yang akan di-inherit oleh class yang baru sering disebut juga sebagai **base** class, sedangkan class baru tadi disebut dengan **derived** class.

9.1.1 Base Class

Inheritance dapat digambarkan sebagai berikut:



Gambar 9-1 Base class dan Derived Class

Jika diwujudkan dalam code, syntax-nya menjadi seperti berikut ini:

Untuk Base class:

```
public class Base
{
    // codes
}
```

Sedangkan untuk Derived class:

```
public class Derived: Base
{
    // codes
}
```

Dalam menulis code untuk derived class, anda menulis tanda titik dua “:” sesudah nama class, kemudian diikuti dengan nama dari base class.

Derived class tadi akan meng-inherit segala hal yang dimiliki oleh base class-nya kecuali **constructor** dan **destructor**-nya. Seluruh **public** member dari base class secara implicit juga menjadi public member dari derived class. Sedangkan **private** member dari base class, meskipun juga di-inherit oleh derived class, hanya dapat diakses oleh base class itu sendiri.

9.1.2 Base Class Member

Sekarang kita akan melihat suatu access modifier yaitu **protected**. Member dari suatu derived class dapat mengakses member **protected** dari base class-nya seperti halnya mengakses member base class yang **public**. Untuk lebih jelasnya, perhatikan contoh berikut:

```
public class Base
{
    protected string nama;
}

public class Derived: Base
{
    public string Nama()
    {
        return nama;
    }
}
```

Akan tetapi, member **protected** dari sebuah class tidak dapat diakses oleh class lain yang tidak derive dari base class tadi.

Contohnya, pada code berikut:

```
public class Other
{
    void Try(Base b)
    {
        return b.nama;
    }
}
```

Class **other** tidak men-derive class **base**, sementara “nama” adalah member **protected** dari class **Base**. Code ini tidak akan berjalan. Anda akan mendapat pesan error berikut saat anda meng-compile:

‘Base.nama’ is inaccessible due to its protection level

Lebih jauh lagi, jika sebuah class meng-inherit sebuah protected member, secara implisit, protected member tadi juga menjadi protected member dari derived class. Akibatnya, protected member dari suatu base class dapat diakses oleh semua class yang derived baik langsung maupun tidak langsung dari base class tersebut. Berikut adalah contoh dari class yang inherit dari class Derived yang kita punya. Class ini tetap dapat mengakses member protected (dalam hal ini "nama") dari base class kita, yaitu Base.

```
public class DerivedMore: Derived
{
    public string NamaJuga()
    {
        return nama + "juga";
    }
}
```

Perlu dicatat bahwa, inheritance hanya berlaku pada class. Anda tidak dapat menerapkan inheritance pada **Struct**. Artinya juga, access modifier protected tidak dapat dipakai di struct.

9.1.3 Base Class Constructor

Sebuah derived class dapat memanggil constructor dari base class-nya. Ini dilakukan dengan menggunakan keyword **base**.

```
public class Token
{
    public Token(string name)
    {
        // codes
    }
}

public class CommentToken: Token
{
    public CommentToken(string name): base(name)
    {
        //codes
    }
}
```

Tanda titik dua yang diikuti constructor dari base class disebut sebagai **constructor initializer**.

Selanjutnya, perhatikan contoh code dengan constructor tanpa parameter berikut:

```
public class Token
{
    public Token()
    {
        // codes
    }
}
```

Untuk memanggil constructor dari base class anda dapat menulis constructor initializer secara eksplisit seperti:

```
public class CommentToken: Token
{
    public CommentToken(): base()
    {
        //codes
    }
}
```

Akan tetapi, jika derived class tidak secara eksplisit memanggil constructor dari base class diatas, C# compiler akan secara implisit memanggil constructor initializer. Contoh code berikut adalah valid:

```
public class CommentToken: Token
{
    public CommentToken() // implisit
    {
        //codes
    }
}
```

Hal ini dimungkinkan karena:

Di dalam .NET, sebuah class yang tidak memiliki base class yang eksplisit akan secara implisit men-derive class **System.Object** yang memiliki constructor tanpa parameter.

Jika sebuah class tidak mempunyai constructor, secara otomatis compiler C# akan membuat sebuah constructor public tanpa parameter yang disebut **default constructor**.

Akan tetapi, jika di suatu class sudah memiliki constructor-nya sendiri, compiler C# tidak akan membuat default constructor. Akibatnya, jika sebuah derived class memiliki constructor yang tidak ada 'pasangan' yang sama di base-class-nya, code anda akan **error**. Contoh:

```
public class Token
{
    public Token(string name)
    {
        // codes
    }
}

public class CommentToken: Token
{
    public CommentToken(string name) // invalid
    {
        //codes
    }
}
```

Error terjadi karena secara implisit, constructor CommentToken memiliki constructor initializer "base()", sementara class Token tidak memiliki constructor tanpa parameter

tersebut. Maka, contoh code diatas harus diubah seperti contoh code di awal sub-bab base constructor ini.

Perlu diperhatikan juga bahwa, access modifier constructor berlaku sama dengan access modifier method biasa lainnya. Artinya, jika sebuah constructor adalah public, dia dapat di akses dari luar class, sedangkan jika suatu constructor adalah private, dia hanya bisa di akses oleh class itu sendiri.

Selanjutnya, coba perhatikan kembali contoh code berikut:

```
public class Base
{
    protected string nama;
}

public class Derived: Base
{
    public string Nama()
    {
        return nama;
    }
}
```

Karena sebuah derived class dapat memiliki member dengan nama yang sama dengan base class, akan lebih baik jika menggunakan keyword **base** untuk mengakses member base class dari suatu derived class. Sehingga contoh code diatas dapat ditulis:

```
public class Derived: Base
{
    public string Nama()
    {
        return base.nama;
    }
}
```

9.2 Implementasi Method

Hal yang menarik dari inheritance adalah anda dapat mendefinisi ulang suatu implementasi method dari sebuah base class jika method base class tersebut didesain untuk dapat di **override**.

9.2.1 Virtual Method

Untuk dapat membuat sebuah method dari suatu base class bisa di override oleh derived class-nya, method tersebut harus diberi keyword **virtual**.

Contoh:

```
public class Base
{
    public virtual string nama()
    {
        // codes
    }
}
```

```
}
```

Jadi, virtual method adalah method yang secara polymorphism dapat di override oleh derived class. Method yang tidak virtual, tidak dapat di derive oleh derived class.

Berikut beberapa hal yang harus diperhatikan:

Sebuah virtual method harus ada implementasinya di base class. Jika tidak ada implementasi, maka program anda akan error.

Sebuah virtual method tidak dapat diberi access modifier **private**. Jika method adalah private, hanya class pemilik method itu sendiri yang dapat mengaksesnya.

Sebuah virtual method juga tidak dapat dideklarasikan sebagai **static**. Polymorphism hanya berlaku pada object, bukan level class. Sementara static member adalah member dari class. Bukan object.

9.2.2 Override Method

Jika pada base class method yang boleh di override harus diberi keyword virtual, maka pada derived class, method yang akan meng-override method base class tadi harus diberi keyword **override**. Contoh:

```
public class Derived : Base
{
    public override string nama()
    {
        // codes
    }
}
```

Beberapa hal yang harus anda perhatikan:

Seperti halnya pada virtual method, pada override method harus juga terdapat implementasi code. Jika tidak, maka program akan error.

Signature dari override method harus sama dengan virtual method yang akan di override.

Seperti pada virtual method, override method juga tidak boleh static, dan tidak boleh private.

Secara implisit, sebuah override method adalah virtual. Tetapi, anda tidak dapat membuat secara eksplisit sebuah override method menjadi virtual. Contoh code berikut adalah invalid:

```
public class Derived : Base
{
    public virtual override string nama() // invalid
    {
        // codes
    }
}
```

9.2.3 Keyword new

Perhatikan contoh code berikut:

```
public class Base
{
    public string Nama()
    {
        // Codes
    }
}

public class Derived: Base
{
    new public string Nama()
    {
        // codes
    }
}
```

Keyword **new** pada method derived class diatas berfungsi untuk menyembunyikan method yang di derived dari base class. Method yang didrived dari base class di ganti oleh method yang memiliki keyword new.

Keyword new dapat menyembunyikan method virtual maupun non-virtual dari base class.

9.3 Sealed Class

Pada banyak kasus, class-class dalam sebuah program adalah sebuah class yang stand alone. Maksudnya, class tersebut tidak dirancang untuk di-derived. Akan tetapi, seorang developer dapat saja menderived sembarang class yang dia inginkan. Padahal class tersebut tidak untuk di-derived.

Untuk mencegah hal ini, di C# ada keyword **sealed**. Sebuah class yang di deklarasi sebagai sealed class tidak dapat di-derived.

Contoh code dari sebuah sealed class:

```
public sealed class Solitaire
{
    // codes
}
```

Class solitaire diatas tadi tidak akan bisa diderive. Maka dari itu, jika anda mencoba untuk menderive class tersebut seperti contoh di bawah ini:

```
public class Derivative: Solitaire // invalid
{
    // codes
}
```

Anda akan mendapat error message berikut:

'Derivative' : cannot inherit from sealed class 'Solitaire'

9.4 Abstract Class

Abstract class digunakan untuk menyediakan sebagian implementasi saja, implementasi lebih jauh akan disediakan oleh concrete class yang men-derived abstract class tersebut. Hal ini akan sangat berguna untuk reusability implementasi suatu interface oleh banyak derived class.

9.4.1 Abstract Class

Abstract class ditandai dengan digunakannya keyword **abstract**. Berikut adalah contohnya:

```
public abstract class Thinker
{
    // codes
}
```

Perlu dicatat bahwa sebuah abstract class tidak dapat di-instantiate. Anda tidak dapat membuat object dari sebuah abstract class. Perhatikan contoh code berikut:

```
public class Try
{
    static void Main()
    {
        Thinker thinker = new Thinker(); // invalid
    }
}
```

Code tersebut akan mendapat compile-time error message sebagai berikut:

```
Cannot create an instance of the abstract class or interface
'Thinker'
```

9.4.2 Abstract Method

Di dalam sebuah abstract class. Kita dapat membuat **abstract** method. Abstract method adalah method yang tidak ada implementasinya. Implementasi dari method semacam ini dilakukan di concrete class yang derived dari abstract class tersebut. Jika anda membuat implementasi code dari sebuah abstract method, maka program anda akan error.

Contoh:

```
public abstract class Base
{
    public abstract string Nama()
    {
        // invalid
        // codes -> // invalid
    }
    // invalid
}
```



```
}
```

Anda seharusnya menulis seperti ini:

```
public abstract class Base
{
    public abstract string Nama();
}
```

Perlu ditekankan disini bahwa hanya class abstract saja yang dapat memiliki abstract method. Sebuah concrete class tidak bisa mempunyai abstract method. Contoh code berikut adalah invalid:

```
public class Base
{
    public abstract string Nama(); // invalid
}
```

Secara implisit, sebuah method abstract adalah juga method virtual. Tetapi anda tidak dapat melakukan secara eksplisit seperti contoh code berikut:

```
public abstract class Base
{
    public virtual abstract string Nama(); // invalid
}
```

Seperti dijelaskan diatas, karena secara implisit sebuah abstract method adalah virtual method, maka contoh code berikut adalah valid, anda dapat meng-override abstract method:

```
public abstract class Base
{
    public abstract string Nama();
}

public class Derived: Base
{
    public override string Nama()
    {
        // codes
    }
}
```

Selanjutnya, sebuah method abstract dapat meng-override method virtual dari suatu base class. Contoh:

```
public class Base
{
    public virtual string Nama()
    {
        // Codes
    }
}

public abstract class Derived: Base
```

```
{
    public abstract override string Nama();
}
```

Terakhir, sebuah abstract method dapat meng-override sebuah override method. Contoh:

```
public class Base
{
    public virtual string Nama()
    {
        // Codes
    }
}

public class Derived: Base
{
    public override string Nama()
    {
        // codes
    }
}

public abstract class FurtherDerived: Derived
{
    public abstract override string Nama();
}
```

9.5 Penutup

Inheritance yang dijelaskan pada bab ini adalah salah satu saja dari dua jenis inheritance yang ada. Bukan. Bukan tentang single ataupun multiple inheritance. Tetapi dua jenis inheritance yaitu **implementation** inheritance dan **interface** inheritance. Bab ini menjelaskan apa yang disebut dengan implementation inheritance. Yaitu inheritance yang dilengkapi dengan implementation. Sedangkan interface inheritance tidak memiliki implementasi. Inheritance-nya lebih kepada interface saja. Bab selanjutnya akan lebih detail membahas tentang interface inheritance atau sering disebut dengan interface saja.

Bab ini juga lebih fokus pada bagaimana menerapkan inheritance sebagai code pada C#. Sintaksis. Anda dapat melanjutkan mencari bacaan tambahan tentang inheritance secara konseptual ataupun bagaimana mendesain class-class yang akan berkolaborasi dengan inheritance di dalam sebuah program. Inheritance adalah sebuah topik yang besar. Tetapi anda sudah memulai mempelajarinya di jalan yang tepat, yaitu menguasai terlebih dahulu sintaksis inheritance di C#.

10. Interface

Fathur Rahman

Dalam OOP yang terpenting adalah apa yang bisa dilakukan oleh object, bukan bagaimana object melakukan sesuatu. Jadi ketika kita bicara object, fokusnya adalah “bisa apa sih?”. Saya punya mesin cuci, bisa apa sih mesin cuci saya? “Mesin cuci dirumah saya hebat, yang kotor bisa bersih sih!” Yang lain menimpali, “Itu mah kecil, dirumah saya baju dimasukkan keluar udah disetrika”, Yang lain lebih hebat, “ Itu enggak ada apa-apanya, datang kerumah saya, masukkan baju ke mesin silahkan mandi begitu keluar kamar mandi, baju seperti baru dan rapi dan udah melakat di badan”. Itu obrolan waktu kita kecil dulu, begitu kita besar, rupanya didalam mesin ada seorang pembantu. Begitulah kira-kira, kita tidak begitu peduli bagaimana mesin itu bekerja yang penting baju kita bersih. Namun ada juga iklan yang menunjukkan keunggulan dengan bagaimana mesin itu bekerja, walupun terkadang tipu belaka. Namanya juga iklan.

Poin penting yang ingin saya sampaikan disini adalah ada semacam tuntutan bahwa suatu object harus memenuhi kemampuan tertentu. Object dikatakan sebagai mesin cuci jika dia mampu mencuci. Siapapun yang pernah mengenal dan memakai mesin cuci dia dengan mudahnya akan mengerti bagaimana menggunakan mesin cuci, apapun mereknya. Dan terkadang juga apapun bahasanya. Ada semacam kesepakatan atau kontrak antar produsen mesin cuci, bahwa mesin cuci mesti mempunyai fungsi minimal seperti ini. Kontrak semacam ini dalam C# disebut sebagai **Interface**. Interface adalah komponen vital dan penting dalam dunia OOP.

10.1 Deklarasi Interface

Berikut ini adalah deklarasi interface yang saya sederhanakan:

```
modifier interface name {  
    return-type method-name1(parameter-list);  
    return-type method-name2(parameter-list);  
    //---  
    return-type method-nameN(parameter-list);  
}
```

Berikut ini contoh interface IMesinCuci:

```
public interface IMesinCuci  
{  
    void SetCuciTimer(Timer timer);  
    void SetInput(Baju[] baju);  
    void SetSabun(Sabun sabun);  
    void Cuci();  
    void SetKerinkanTimer(Timer timer);  
}
```

```
        void Keringkan();  
        Baju[] GetBaju();  
    }
```

Huruf pertama dari nama interface kita, I, ini hanya masalah penamaan saja. Jika anda menemui class yang diawali dengan I hamper pasti ini adalah interface. Hampir pasti he..he..Anda mesti pastikan. Dalam interface tidak boleh ada implementasi method. Dan semua method adalah public. Semua class atau struct yang mengimplementasikan interface ini harus mengimplementasikan semua method tanpa kecuali. Dan tentunya dilarang keras membuat method **static** disini.

Selain method interface juga menerima, property, indexer dan event. Saat ini saya tidak akan memasukkan ke tiga bahasan ini. Saya serahkan pada pembaca untuk meneruskannya, kecuali anda memaksa.

10.2 Implementasi Interface

Berikut adalah deklarasi implementasi *class* yang memakai interface yang saya sederhanakan:

```
class class-name: interface-name1, interface-name2{  
    //class-body  
}
```

Class yang mengimplementasikan interface wajib mengimplementasikan semua method secara public. Interface yang diimplementasikan boleh lebih dari satu yang dipisahkan oleh koma.

Berikut ini adalah contoh implemantasi interface mesin cuci:

```
using System;  
public interface IMesinCuci  
{  
    void SetCuciTimer(Timer timer);  
    void SetInput(params Baju[] bajus);  
    void SetSabun(Sabun sabun);  
    void Cuci();  
    void SetKeringkanTimer(Timer timer);  
    void Keringkan();  
    Baju[] GetBaju();  
}  
  
public class Timer  
{  
    public Timer(int val)  
    {  
        _timer=val;  
    }  
    public int Value  
    {  
        get{return _timer;}  
    }  
    private int _timer;  
}  
  
public class Baju  
{
```

```

        public Baju(string name)
        {
            _name=name;
        }
        public string Name
        {
            get{return _name;}
        }
        private string _name;
        public override string ToString()
        {
            return "Test" + _name;
        }
    }

    public class Sabun
    {
        public Sabun(string name)
        {
            _name=name;
        }
        public string Name
        {
            get{return _name;}
        }
        private string _name;
    }

    public class MCMerekFatur: IMesinCuci
    {
        private string _logo;
        private Timer _cuciTimer;
        private Timer _keringkanTimer;
        private Baju[] _drum;
        private Sabun _sabun;
        public MCMerekFatur()
        {
            _logo="Fatur Oye";
        }
        public string GetLogo()
        {
            return this._logo + " (Ngebling he..he..)";
        }
        #region implementasi IMesinCuci
        public void SetCuciTimer(Timer timer)
        {
            _cuciTimer=timer;
        }
        public void SetKeringkanTimer(Timer timer)
        {
            _keringkanTimer=timer;
        }
        public void SetInput(params Baju[] bajus)
        {
            _drum=bajus;
        }
        public void SetSabun(Sabun sabun)
        {
            _sabun=sabun;
        }
        public void Cuci ()
        {
            Console.WriteLine("Anda Sedang Menggunakan {0} ",
                this.GetLogo());
        }
    }

```

```

        Console.WriteLine("Masukkan sabun {0} ",
            _sabun.Name);
        Console.WriteLine("Mulai mencuci i i...");

        for(int i=1; i<=_cuciTimer.Value; i++)
        {
            Console.WriteLine("Putaran ke {0} ", i);
        }
        Console.WriteLine(
            "Priiiiit, Priiiiit mencuci selesai");
        Console.WriteLine(
            "Coba check j angan-j angan gosong");
    }
    public void Keringkan()
    {
        Console.WriteLine("Mulai mengeringkan...");
        for(int i=1; i<=_keringkanTimer.Value; i++)
        {
            Console.WriteLine("Putaran ke {0} ", i);
        }
        Console.WriteLine(
            "Priiiiit, Priiiiit mengeringkan selesai");
        Console.WriteLine(
            "Aduhh... anda lupa kasih air ya tadi!!#$$%^*");
    }
    public Baju[] GetBaju()
    {
        Console.WriteLine("Ngiiii ng buka tutup");
        for(int i=0; i<_drum.Length; i++)
        {
            //Baju b =_drum[i] as Baju;
            Console.WriteLine("Keluarkan {0}"
                , _drum[i].Name);
        }

        Console.WriteLine("Nguiii nng Tutup Kembali");
        Console.WriteLine(
            "Terai makasih Anda Telah Menggunakan {0} ",
            this.GetLogo());
        return _drum;
    }
}
#endregion

public class MCMerekRahman: IMesinCuci
{
    private string _name;
    private Timer _cuciTimer;
    private Timer _keringkanTimer;
    private Baju[] _drum;
    private Sabun _sabun;
    public MCMerekRahman()
    {
        _name="Rahman Washing Machine";
    }
    public string Name
    {
        get{
            return this._name;
        }
    }
}
#region implementasi IMesinCuci
public void SetCuciTimer(Timer timer)

```

```

    {
        if(timer.Value<0 || timer.Value>10)
            timer=new Timer(10);

        _cuciTimer=timer;
    }
    public void SetKeringkanTimer(Timer timer)
    {
        if(timer.Value<0 || timer.Value>5) timer=new Timer(5);
        _keringkanTimer=timer;
    }
    public void SetInput(params Baju[] bajus)
    {
        _drum=bajus;
    }
    public void SetSabun(Sabun sabun)
    {
        _sabun=sabun;
    }
    public void Cuci ()
    {
        Console.WriteLine("Anda Sedang Menggunakan {0} ",
            thi s. Name);
        Console.WriteLine("Sedot air.....");
        Console.WriteLine("Masukkan sabun {0} ",
            _sabun. Name);
        Console.WriteLine("Mul ai mencuci i i....");
        for(int i=1; i<=_cuciTimer. Value; i++)
        {
            Console.WriteLine("Putaran ke {0} ke kanan", i);
            Console.WriteLine("Putaran ke {0} ke kiri ", i);
            Console.WriteLine("Putaran ke {0} sedot ke bawah", i);
        }
        Console.WriteLine("Mencuci sel esai");
        Console.WriteLine("Kel uarkan Air");
        Console.WriteLine("Masukkan Air");
        Console.WriteLine("Putar Kembali");
        Console.WriteLine("Kel uarkan Air");
    }
    public void Keringkan()
    {
        Console.WriteLine("Mul ai mengeri ngkan....");
        for(int i=1; i<=_keringkanTimer. Value; i++)
        {
            Console.WriteLine("Putar dengan kecepatan {0} ", i);
        }

        Console.WriteLine("Pengeri ngan sel esai");
    }
    public Baju[] GetBaju()
    {
        Console.WriteLine("Buka penutup");
        for(int i=0; i<_drum. Length; i++)
        {
            //Baju b =_drum[i] as Baju;
            Console.WriteLine("Kel uarkan {0}", _drum[i]. Name);
        }

        Console.WriteLine("Kembali kan Penutup");
        Console.WriteLine(
            "Terima kasih anda telah menggunakan {0}",
            thi s. Name);
        return _drum;
    }

```

```

    }
    #endregion
}

public class Pembantu
{
    private string _name;
    private object[] _keranjang;
    public Pembantu(string name)
    {
        _name=name;
    }
    public string Name
    {
        get{return _name;}
    }
    public void Mencuci (IMesi nCuci mc,
                        Sabun sabun, params Baju[] baju)
    {
        mc.SetSabun(sabun);
        mc.SetInput(baju);
        mc.SetCuciTimer(new Timer(20));
        mc.Cuci();
        mc.SetKeringkanTimer(new Timer(15));
        mc.Keringkan();
        _keranjang=mc.GetBaju();
    }
    public void Menjemur()
    {
        Console.WriteLine(
            "{0} menjemur baju sebanyak {1}
            buah.",
            this.Name, _keranjang.Length);
        foreach(Baju b in _keranjang)
        {
            Console.WriteLine("---Menjemur {0}...", b.Name);
        }
        Console.WriteLine("Selasai menjemur");
    }
}

public class UnitTest
{
    public static void Main()
    {
        MCMerekFatur mcf=new MCMerekFatur();
        MCMerekRahman mcr =new MCMerekRahman();
        Baju celana=new Baju("Celana");
        Baju kaos=new Baju("kaos");
        Baju hem=new Baju("hem");
        Baju cel dam=new Baju("Cel dam");
        Baju kaoskaki=new Baju("Kaos Kaki");
        Baju kutang =new Baju("Kutang");
        Sabun ri nso=new Sabun("Ri nso");
        Pembantu inem=new Pembantu("Inem");
        inem.Mencuci (mcf, ri nso, cel ana, kaos,
                    hem, cel dam, kaoskaki , kutang);
        inem.Menj emur();
        Sabun attack=new Sabun("Attack+B29");
        inem.Mencuci (mcr, attack, cel ana, kaos, hem);
        inem.Menj emur();
    }
}

```



```
}
```

Kita buat interface Mesin Cuci.

Kemudian kita buat *class* mesin cuci yang mengimplementasikan interface mesin cuci. Bagian ini, antara **#region** dan **#endregion** adalah definisi seluruh kontrak karya yang harus dibuat karena mengimplementasikan interface.

Kita buat lagi *class* mesin cuci yang lain, yang berbeda cara kerjanya. Perhatikan bagaimana *class* mengimplementasikan interface. Disini waktu mencucinya dibatasi begitu juga waktu mengeringkannya. Waktu mencuci kelihatan lebih cerdas. Cerdas? He..he..

Sama seperti bagian no 3. implementsi dari interface yang ditanda-tangani. Tidak peduli implementasinya seperti apa.

Perhatikan bagaimana *class* pembantu merujuk ke *class* mesin cuci. Pembantu cukup memakai IMesinCuci. Pokoknya pembantu tinggal set ini set itu, tekan tombol ini tekan tombol itu, apapun mesinnya. Pembantu tak peduli bagaiman kerja didalamnya. Mau pake mesin kek, mau pake orang dalam mesin kek. Terserah, bebas.

Kita beli mesin cuci merek fatur.

Kita beli juga mesin cuci merek rahman. Semua baju kita siapkan. Pembantu kita panggil.

Pembantu mencuci kemudian menjemur.

Perhatikan, inem hanya cukup tahu interfacenya. Inem hanya perlu dikasih tahu bahwa mesin cuci itu kerjanya gini dan gini lho. Apapun mesinnya Inem tahu bagaiman menggunakannya.

Di contoh ini ada beberapa bagian yang diluar nalar, misalnya, ketika pakaian diambil keluar mestinya pakaian dalam mesin cuci sudah tidak ada. Disini masih ada. Misalnya lagi, baju-baju udah dicuci dan dijemur kok bisa dicuci lagi. Tuannya atau inemnya yang gak cerdas! Ini saya serahkan kepada pembaca, bagaimana mengatasinya supaya domain ini bisa lebih sempurna. Inilah enakunya jadi penulis, contohnya enggak perlu sempurna. Berbeda dengan programmer atau developer (bukan developer rumah lho he..he..), mereka harus lebih serius, harus sempurna, tidak boleh ada bug sedikit pun.

.Net Framwork banyak memakai konsep interface ini dan anda akan senang menggunakannya dan lama-lama akan terbiasa. Curigailah semua yang berawalan degan I di .Net framwork sebagai Interface. Kemudian carilah bagaimana mengimplementasikannya. Saya sisakan untuk anda pelajari sendiri Interface Properties, Interface Indexer dan Interface Event, kecuali anda memaksa. Juga saya tidak akan menjelaskan apa perbedaannya dengan *abstract class*, karena sudah jelas dan ada penampakan. Misal, turunan *abstract class / class* hanya boleh satu sedangkan interface boleh banyak. Misal lagi, *abstract class* method yang belum sempurna harus diawali dengan modifier *abstract* sedangkan interface tidak. Penampakan-penampakan yang lain? Anda harus mencoba baru bisa merasakan konsekuensinya.

Sekali lagi terimakasih.

11. Collection

I Wayan Saryada

11.1 Array

Di bagian 3 kita memperkenalkan penggunaan variabel dalam C#, suatu variabel akan dapat menampung satu data pada suatu saat. Ada kalanya kita menginginkan untuk menyimpan sekelompok data yang bertipe sama ke dalam satu variabel. Sebagai contoh kita ingin menyimpan nama-nama bulan, kita bisa saja membuat 12 buah variabel untuk menyimpan masing-masing nama bulan, tetapi alangkah baiknya misalnya jika kita dapat menggunakan satu buah variabel untuk menyimpan ke 12 buah nama bulan ini.

Mungkin contoh ini tidak terlalu ekstrim, bisa saja kita membuat 12 nama variabel yang berbeda dan ini bukan sesuatu yang sulit, tapi misalnya jika kita ingin menyimpan 1000 bilangan prima pertama, maka bisa dipastikan kita tidak akan membuat 1000 buah variabel untuk masing-masing bilangan prima, karena ini akan sangat tidak efisien dalam program.

Penggunaan array dapat memenuhi kebutuhan ini, array adalah suatu struktur data yang dapat menyimpan data dengan tipe yang sama dan diakses dengan menggunakan suatu indeks yang menunjukkan suatu elemen didalam array tersebut. Variabel yang disimpan didalam array ini disebut juga dengan elemen array dan tipe datanya disebut dengan tipe elemen dari array.

11.1.1 Mendeklarasikan Array

Adapun ekspresi yang digunakan untuk mendeklarasikan array adalah:

```
type[] name;
```

Type adalah tipe data dari array dan *name* adalah nama dari array, sedangkan tanda [] memberitahu C# untuk membuat variabel array. Tidak seperti bahasa lainnya dalam C# tanda [] harus diletakan setelah tipe data.

```
string[] namaBulan;
```

Pada contoh diatas kita membuat sebuah variabel array bernama `namaBulan` data bertipe `string`.

11.1.2 Meng-Initialize Array

Untuk memberikan nilai ke dalam variabel array dapat dilakukan dengan beberapa cara. Pertama dengan menggunakan keyword `new` untuk menentukan jumlah element yang dapat disimpan oleh array yang dilanjutkan dengan memberikan nilai masing-masing element.

```
string[] namaBulan = new string[12];
namaBulan[0] = "Januari";
namaBulan[1] = "Pebruari";
namaBulan[12] = "Desember";
```

Kedua dengan cara diinitialize dengan langsung memberi nilai pada saat deklarasi seperti ditunjukkan pada contoh berikut:

```
string[] namaBulan = {"Januari", "Pebruari", "...", "Desember"};

string[] namaBulan = new string[] {"Januari", "Pebruari", "...",
"Desember"};
```

Jika menggunakan dua cara diatas ini maka jumlah elemen array ditentukan berdasarkan array initializer yang diberikan.

11.1.3 Mengakses Array

Untuk mengakses data yang tersimpan di dalam array digunakan operator `[]`. Array dalam C# indeksnya dimulai dari 0 sampai N-1, dimana N adalah jumlah elemen yang dideklarasikan. Jadi untuk mengakses data yang pertama didalam `namaBulan` diatas kita dapat menggunakan ekspresi `namaBulan[0]`, untuk mengakses elemen kedua adalah `namaBulan[1]` dan seterusnya. Sebagai indeks dapat digunakan bilang bulat bertipe `int`, `uint`, `long`, `ulong` atau tipe data lain yang bisa dikonversi menjadi tipe data ini.

Array tidak boleh diakses pada indeks yang ada diluar indeks array yang valid yaitu 0 – N-1. Jika hal ini dilakukan akan terjadi exception pada saat runtime berupa `System.IndexOutOfRangeException`.

11.1.4 Contoh Program

Program berikut membuat array dengan mendeklarasikannya dan memberikan nilai. Kemudian program akan menampilkan jumlah elemen yang ada didalam array dan dilanjutkan dengan menampilkan semua elamen didalamnya.

```
using System;

public class MyArray
{
    public static void Main()
    {
        string[] namaBulan = { "Januari", "Pebruari", "Maret",
                                "April", "Mei", "Juni", "Juli",
                                "Agustus", "September", "Oktober",
                                "Nopember", "Desember" };

        Console.WriteLine("Jumlah elemen = {0}",
            namaBulan.Length);
    }
}
```

```
        for (int i = 0; i < namaBulan.Length; i++)
        {
            Console.WriteLine("Bulan {0} = {1}", i+1,
                               namaBulan[i]);
        }
    }
```

11.2 Collection

Collection, seperti tergambar dari namanya berfungsi sebagai kontainer untuk menyimpan sekumpulan object, collection hampir mirip dengan konsep array. Untuk mengakses data didalam collection biasanya dilakukan dengan melakukan iterasi terhadap data yang ada didalam collection atau dapat juga diakses dengan menggunakan indeks atau *indexers*.

.NET Framework membedakan collection menjadi dua macam yaitu yang berfungsi sebagai *general purposes* collection dan *specific purposes* collection. Namespace System.Collections berisikan sekumpulan class dan interface yang membentuk collection framework ini, seperti ICollection, IEnumerable, IDictionary, IList, Hashtable, ArrayList. Dengan tersedianya interface-interface ini kita akan dapat membuat suatu implementasi collection yang baru. Sedangkan namespace System.Collections.Specialized berisikan collection untuk menangani hal-hal yang bersifat spesifik seperti StringCollection dan StringDictionary.

Berikut akan coba dibahas satu persatu mengenai initerface-interface yang disebutkan diatas.

11.2.1 ICollection

ICollection adalah interface utama dari semua class collection, interface ini diimplementasikan oleh semua class collection didalam .NET Framework. Semua implementasi collection paling tidak harus mengimplementasikan interface ICollection yang merupakan syarat minimal untuk membuat class collection. Adapun deklarasi dari ICollection adalah sebagai berikut:

```
interface ICollection
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo(Array array, int index);
}
```

Adapun fungsi dari masing-masing member dari interface ICollection ini adalah sebagai berikut:

Count menyimpan jumlah object yang terdapat didalam collection.

CopyTo() berfungsi untuk mengkopi collection kedalam bentuk array.

IsSynchronized akan memberikan nilai true jika collection bersifat *thread-safe* dan false jika sebaliknya.

SyncRoot akan memberikan object sinkronisasi dalam menggunakan collection dalam konteks multi-threaded.

Properti `IsSynchronized` dan `SynchRoot` berfungsi untuk mendukung konsep dasar pemrograman multi-threading.

11.2.2 IEnumerable

Untuk melakukan iterasi dengan menggunakan statemen `foreach` collection harus mengimplementasikan interface `IEnumerable`. Adapun deklarasi dari `IEnumerable` adalah sebagai berikut:

```
interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Interface ini hanya memiliki satu member yaitu `GetEnumerator()` yang menghasilkan object `IEnumerator` yang memiliki kemampuan untuk melakukan iterasi terhadap collection. `IEnumerator` hanya mengijinkan membaca data didalam collection tetapi tidak untuk mengubahnya.

11.2.3 IList

`IList` merepresentasikan collection yang dapat diakses dengan menggunakan indexer. `IList` merupakan turunan dari interface `ICollection` dan merupakan base interface dari semua collection bertipe list. Adapun deklarasi dari interface `IList` ini adalah sebagai berikut:

```
interface IList
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[object key] { get; set; }
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object key);
    void RemoveAt(int index);
}
```

Contoh dari class collection yang mengimplementasikan interface ini adalah `ArrayList` dan `StringCollection`.

11.2.4 IDictionary

`IDictionary` adalah collection yang merupakan pasangan *key-value*. `IDictionary` adalah interface utama dari semua collection bertipe dictionary. Setiap elemen disimpan sebagai *key-value* dalam object `DictionaryEntry`. Key dari elemen collection ini tidak boleh berupa object `null`, tetapi boleh berisi value bernilai `null`.

Deklarasi dari interface `IDictionary` adalah:

```
interface IDictionary
```

```
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    ICollection Keys { get; }
    ICollection Values { get; }
    object this[object key] { get; set; }
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);
}
```

`IDictionary` memiliki method `GetEnumerator()` yang menghasilkan `IDictionaryEnumerator`, interface ini adalah turunan dari `IEnumerator` yang khusus digunakan dalam melakukan iterasi dalam collection bertipe dictionary.

`IDictionary` bisa dicontohkan penggunaannya seperti sebuah kamus, dimana didalam kamus akan berisikan data yang berupa pasangan antara kata dengan artinya. Setiap kata (key) dalam kamus akan memiliki suatu arti (value) tertentu.

11.2.5 Queue

Queue mengimplementasikan tiga interface yaitu `ICollection`, `IEnumerable` dan `ICollection`. Queue menyimpan data secara FIFO (first-in first-out), dalam struktur data ini data yang disimpan pertama akan dibaca pertama juga.

Queue secara internal akan menyimpan data dalam sebuah buffer berupa array. Jika batas maksimal data yang dapat disimpan queue sudah tercapai maka secara otomatis queue akan mengalokasikan buffer baru dengan cara membuat sebuah array baru yang dan kemudian meng-copy data di buffer yang lama ke buffer yang baru.

Jika menyimpan data dalam jumlah yang besar proses pengalokasian ulang buffer ini dapat menurunkan performance program kita, maka untuk mencegahnya kita dapat mendeklarasikan queue dengan kapasitas awal. Contoh pendeklarasiannya adalah sebagai berikut:

```
Queue myQueue = new Queue(20);
```

Selain menentukan kapasitas awal kita juga dapat menentukan faktor untuk penambahan jumlah buffer didalam queue, ini disebut dengan *growth factor*. Dengan menentukan *growth factor* ini secara tepat maka kita akan dapat mengurangi beban pengalokasian buffer pada queue yang kita buat.

```
Queue myQueue = new Queue(50, 3.0);
```

Berikut ini adalah sebuah contoh sederhana penggunaan Queue.

```
using System;
using System.Collections;

public class MyQueue
{
    public static void Main()
```

```
    {
        Queue queue = new Queue();
        queue.Enqueue("Hello");
        queue.Enqueue("World");

        Console.WriteLine(queue.Peek());
        Console.WriteLine(queue.Dequeue());
        Console.WriteLine(queue.Dequeue());
    }
}
```

Hasilnya adalah:

```
Hello
Hello
World
```

Pertama kita membuat sebuah object queue, kemudian kita memasukan dua buah data yaitu "Hello" dan "World" dengan menggunakan method `Enqueue()`. Untuk membaca data dari dalam Queue kita bisa menggunakan method `Peek()` atau `Dequeue()`. Perbedaan kedua method ini adalah method `Peek` akan mengambil data dari queue tanpa menghapusnya dari queue, sedangkan `Dequeue` akan menghapus data dari queue.

11.2.6 Stack

Stack beroperasi dalam bentuk LIFO (last-in first-out), dalam struktur data ini data yang dimasukan terakhir akan dapat dibaca pertama. Stack juga mengimplementasikan interface `ICollection`, `IEnumerable` dan `ICollection`.

Sama halnya dengan Queue, struktur data internal dari stack adalah array. Untuk mencegah terjadi performance cost yang terlalu besar yang terjadi karena adanya proses pengalokasian ulang buffer ini maka kita juga dapat menentukan ukuran awal dari Stack ini. Adapun cara mendeklarasikan stack dengan ukuran awal ini adalah:

```
Stack stack = new Stack(20);
```

Program berikut memberi contoh penggunaan Stack.

```
using System;
using System.Collections;

public class MyStack
{
    public static void Main()
    {
        Stack stack = new Stack();
        stack.Push("!!");
        stack.Push("World");
        stack.Push("Hello");

        Console.WriteLine("{0} {1} {2}",
            stack.Pop(), stack.Pop(), stack.Pop());
    }
}
```

Hasilnya adalah:

```
Hel lo Worl d !!
```

Pada program diatas kita mulai dengan mendeklarasikan sebuah object Stack, kemudian memasukan tiga buah string secara berurutan. Data dimasukkan dengan urutan "!!", "World", "Hello", kemudian data ini dibaca sebanyak tiga kali. Karena stack memiliki struktur data LIFO maka sewaktu data dibaca dari stack yang dihasilkan adalah kebalikannya menjadi "Hello", "World", "!!".

11.2.7 ArrayList

Jika kita menggunakan array maka kita harus tahu jumlah data yang disimpan di dalam array tersebut, ini adalah salah satu kelemahan dari array. Bagaimana jika kita ingin menggunakan array untuk menyimpan data yang jumlahnya bisa bertambah atau berkurang secara dinamis. Untuk melakukan ini kita dapat menggunakan `ArrayList`, yang merupakan implementasi dari `ICollection`.

Berikut adalah contoh penggunaan `ArrayList` dalam program:

```
using System;
using System.Collections;

public class MyArrayList
{
    public static void Main()
    {
        ArrayList arrayList = new ArrayList();
        for (int i = 1; i <= 10; i++)
        {
            arrayList.Add(i * i);
        }
        Console.WriteLine("Jumlah elemen = {0}",
            arrayList.Count);

        foreach(int i in arrayList)
        {
            Console.WriteLine("{0} ", i);
        }
    }
}
```

11.2.8 StringCollection

`StringCollection` adalah salah satu collection yang digunakan khusus untuk menyimpan data bertipe string. Keuntungan menggunakan collection ini adalah tidak diperlukan untuk melakukan konversi data dari object menjadi string, karena collection secara umum menyimpan dan membaca data bertipe object. `StringCollection` merupakan implementasi dari `ICollection`.

Berikut adalah contoh penggunaan `StringCollection`.

```
using System;
```



```
using System.Collections.Specialized;

public class MyStringCollection
{
    public static void Main()
    {
        StringCollection sc = new StringCollection();
        sc.Add("Hejlsberg");
        sc.Add("Golde");

        sc.Insert(1, "Wilamuth");
        foreach(string name in sc)
        {
            Console.WriteLine(name);
        }
    }
}
```

Hasilnya adalah:

```
Hejlsberg
Wilamuth
Golde
```

11.2.9 Hashtable

Hashtable adalah collection yang bertipe dictionary karena dia mengimplementasikan interface `IDictionary`. Key yang boleh digunakan untuk menyimpan data dalam object ini tidak boleh berupa `null` dan harus merupakan object yang mengimplementasikan method `Object.GetHashCode()` dan `Object.Equals()`.

Didalam Hashtable object akan disimpan sebagai pasangan *key-value* bertipe `DictionaryEntry`. Hashtable dapat menyimpan semua data bertipe object. Untuk membaca dari Hashtable perlu dilakukan *casting* secara explicit. Implementasi khusus Hashtable untuk menyimpan data string adalah class `StringDictionary`.

Contoh penggunaan Hashtable:

```
using System;
using System.Collections;

public class MyHashtable
{
    public static void Main()
    {
        Hashtable production = new Hashtable();
        production.Add("Q1", 10000);
        production.Add("Q2", 15000);
        production.Add("Q3", 17000);
        production.Add("Q4", 25000);

        IDictionaryEnumerator enumerator =
            production.GetEnumerator();
        while (enumerator.MoveNext())
        {
            Console.WriteLine("{0}: {1}",
                enumerator.Key, enumerator.Value);
        }
    }
}
```

```
}
```

Hasilnya adalah:

```
Q3: 17000
Q2: 15000
Q1: 10000
Q4: 25000
```

11.2.10 StringDictionary

StringDictionary tidak mengimplementasikan interface ICollection ataupun IDictionary. StringDictionary bisa dikatakan sebagai Hashtable untuk menyimpan data string, sehingga tidak diperlukan *casting* secara eksplisit untuk mengubah data object ke string seperti layaknya sebuah Hashtable.

Berikut adalah contoh penggunaan StringDictionary:

```
using System;
using System.Collections;
using System.Collections.Specialized;

public class MyStringDictionary
{
    public static void Main()
    {
        StringDictionary sd = new StringDictionary();
        sd.Add("DPS", "Denpasar");
        sd.Add("SBY", "Surabaya");
        sd.Add("JKT", "Jakarta");

        foreach(string key in sd.Keys)
        {
            Console.WriteLine("{0} - {1}", key, sd[key]);
        }
    }
}
```

Hasilnya adalah:

```
sby - Surabaya
dps - Denpasar
jkt - Jakarta
```

Key dari StringDictionary tidak *case-sensitive*, artinya jika kita mengisi StringDictionary dengan data "dps" dan "DPS" akan menyebabkan terjadi kesalahan, karena data ini dianggap sama. Selain itu sewaktu data dimasukkan ke dalam StringDictionary key akan diubah menjadi huruf kecil, seperti terlihat pada hasil program diatas.

12. Namespace

Risman Adnan

12.1. Definisi Namespace

Namespace adalah kata kunci yang digunakan untuk mendefinisikan suatu ruang lingkup atau batasan dalam kode program. *Namespace* berguna untuk mengorganisasikan kode program dan dapat digunakan untuk membuat type data yang global sekaligus unique. Bentuk umum penggunaan *namespace* dalam C# adalah:

```
namespace nama[.nama1] ...  
{  
    deklarasi type;  
}
```

dimana *nama* dan *nama1* adalah nama namespace. Type yang dapat di deklarasikan dalam *namespace* meliputi :

namespace yang lain

- class
- interface
- struct
- enum
- delegate

Namespace merupakan konsep yang dipinjam dari C++ yang dapat digunakan untuk menjamin bahwa semua penamaan yang anda gunakan dalam program bersifat *unique*. Dengan menggunakan namespace, anda bisa mengelompokkan *class*, *interface*, *struct*, *enum* dan *delegate* dengan namespace yang berbeda agar tidak terjadi konflik penamaan dan organisasi dari kode anda menjadi lebih terstruktur. Untuk developer yang bekerja membuat class library dalam proyek software berskala besar, *namespace* penting untuk mengorganisasikan class-class ke dalam satu struktur hirarki.

12.2. Namespace dan Using

Program C# diorganisasikan oleh *namespace*. Namespace dapat digunakan sebagai sistem organisasi “internal” dalam program anda dan juga secara “eksternal” karena dapat diakses secara *public* menggunakan kata kunci **using**. Mari kita membuat program sederhana untuk menampilkan kata “Selamat Bergabung dengan INDC” di layar *console*. Program ini terdiri dari dua project di Visual Studio .NET, komponen **HelloINDC** yang menghasilkan pesan dan satu aplikasi console **HelloConsole** untuk menampilkan pesan.

Pertama-tama, kita mengorganisasikan class HelloINDC ke dalam satu *namespace* sesuai dengan nama organisasi, **INDC.CSharp.Otak**

```
namespace INDC. CSharp. Otak
{
    public class HelloINDC
    {
        public string GetMessage()
        {
            return "Selamat Bergabung Dengan INDC";
        }
    }
}
```

Bentuk diatas adalah penyederhanaan dari bentuk :

```
namespace INDC
{
    namespace CSharp
    {
        namespace Otak
        {
            // class HelloINDC
        }
    }
}
```

Selanjutnya, kita buat aplikasi console yang menggunakan class HelloINDC (lihat bagian 12.4 untuk penambahan referensi). Kita bisa gunakan nama lengkap dari class INDC.CSharp.Otak.HelloINDC di dalam kode program atau menyingkatnya dengan kata kunci **using INDC.CSharp.Otak** di awal kode program.

```
using INDC. CSharp. Otak;
using System;
namespace INDC. HelloConsole
{
    class HelloConsole
    {
        static void Main(string[] args)
        {
            HelloINDC m = new HelloINDC();
            Console.WriteLine(m.GetMessage());
            Console.ReadLine();
        }
    }
}
```

C# juga memungkinkan kita membuat definisi tambahan dan alias untuk penulisan *namespace*. Alias ini dapat berguna dalam situasi dimana terjadi konflik penamaan ketika anda menggunakan banyak library dalam program anda. Untuk contoh di atas kita bisa menulis alias **Hello**:

```
using Hello = INDC. CSharp. Otak. HelloINDC;
using System;
namespace INDC. HelloConsole
{
    class HelloConsole
    {
        static void Main(string[] args)
        {
            Hello m = new Hello ();
        }
    }
}
```

```
        Console.WriteLine(m. GetMessage());
        Console.ReadLine();
    }
}
```

Pada contoh di atas anda juga melihat acuan ke **System** library, yang merupakan library utama dalam .NET Framework. Acuan ke System library diperlukan karena kita membutuhkan *method* untuk penulisan ke console. *Method* tersebut ada di dalam class **System.Console**.

Dalam .NET Framework terdapat lebih dari 90 *namespace* yang berawal dengan kata System sehingga penggunaan namespace menjadi sangat penting. Misalnya, anda ingin membuat class dengan nama *Timer* yang telah digunakan dalam .NET Framework. Class *Timer* juga dapat ditemukan di **System.Timers**, **System.Threading** dan **System.Windows.Forms**. Dengan bantuan namespace dan alias, anda tetap dapat membuat class dengan nama *Timer* selama namespace yang anda gunakan bersifat *unique*.

12.3. Namespace Bertingkat

Namespace juga dapat digunakan secara bertingkat seperti pada contoh berikut:

```
using System;
namespace INDC. HelloConsole
{
    class HelloConsole
    {
        static void Main(string[] args)
        {
            Console.WriteLine(
                Bertingkat. HelloConsole. GetMessage());
            Console.ReadLine();
        }
    }

    namespace Bertingkat // namespace bertingkat
    {
        public class HelloConsole
        {
            public static string GetMessage()
            {
                return "Namespace Bertingkat";
            }
        }
    }
}
```

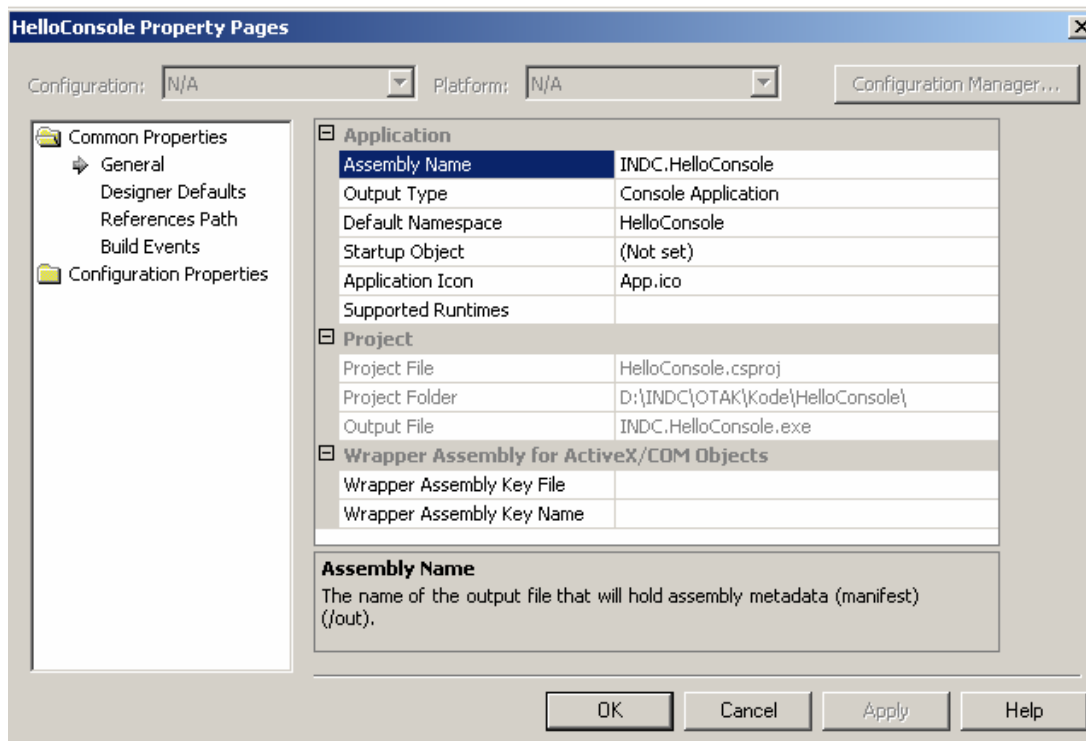
Walaupun tidak menimbulkan error saat kompilasi, penggunaan *namespace* bertingkat tidak disarankan dalam pembuatan class library.

12.4. Namespace dan Assembly

Suatu obyek dapat digunakan dalam kode C# anda hanya jika obyek tersebut dapat ditemukan oleh Compiler yang anda gunakan. Secara default, compiler hanya akan membuka satu assembly yaitu mscorlib.dll, yang berisi fungsi-fungsi utama dalam CLR (*Common Language Runtime*). Untuk memberi acuan kepada obyek-obyek ada di assembly yang lain, anda harus memberi informasi kepada compiler nama dan lokasi assembly tersebut.

Pada contoh 12-2. **HelloINDC** adalah komponen yang digunakan oleh aplikasi **HelloConsole**. Agar HelloConsole dapat memanggil method GetMessage() dalam assembly HelloINDC, anda perlu memberi opsi `/r:<HelloINDC.dll>` ketika melakukan kompilasi aplikasi HelloConsole melalui *command line*. Jika anda menggunakan Visual Studio .NET, penambahan referensi dapat dilakukan melalui menu **Project** → **Add Reference**. Anda dapat menambah referensi ke project HelloINDC ataupun assembly filenya (HelloINDC.dll).

Ketika anda memulai *project* dengan Visual Studio .NET, *namespace* yang diberikan secara *default* sama dengan nama *project*. Selanjutnya, jika anda menambahkan class ke dalam project tersebut, class-class yang baru akan memiliki *namespace* yang sama kecuali jika anda merubah *default namespace* melalui *project properties* yang ada di **Menu Project**. Nama file output (*assembly*) yang dihasilkan dari *project* setelah di compile dapat di set melalui project properties. Gambar berikut ini menunjukkan *project properties* dari **HelloConsole**



Gambar 12-1. Halaman Property dari project console HelloConsole

Nama *assembly* adalah *property* yang menunjukkan nama dari *assembly* (dll atau exe) yang dihasilkan. Nama *assembly* ini sama dengan nama file keluaran *project* (*Output File*). Jika anda merubah nama *assembly*, Visual Studio .NET akan merubah Output file *project* anda secara otomatis. *Default Namespace* adalah nama *namespace* yang akan digunakan oleh Visual Studio .NET saat anda membuat class baru. Ini akan memudahkan anda untuk tetap memiliki satu *unique namespace* untuk tiap *project* yang ada buat. *Default namespace* tidak berhubungan dengan nama *assembly* sehingga anda bebas memberikan nama *default namespace* untuk memudahkan penulisan kode program.

13. File

Adi Wirasta

13.1 Pengenalan Namespace System.IO

Namespace IO berisi tipe yang dapat membaca dan menulis ke file atau data stream. Hirarki namespace IO itu terdiri dari class, structures, delegates, enumerations. Namun yang akan dibahas adalah class saja dan itu hanya sebagian.

Namespace hierarchy
Classes

Class	Description
BinaryReader	Membaca tipe data primitif seperti nilai-nilai binari.
BinaryWriter	Menulis tipe data primitif/binari ke dalam stream.
BufferedStream	Menulis dan membaca ke stream yang lain. Class ini tidak bisa di inherited.
Directory	Merupakan static method untuk membuat, memindahkan dan merunutkan direktori dan subdirektori.
DirectoryInfo	Merupakan non-static method untuk membuat, memindahkan dan merunutkan direktori dan subdirektori
DirectoryNotFoundException	Exception ketika file atau direktori tidak ditemukan.
EndOfStreamException	Exception ketika pembacaan sudah melewati akhir dari stream. Biasanya muncul ketika diakhir baris masih terdapat satu baris dibawahnya berupa baris kosong.
ErrorEventArgs	menyediakan data untuk event Error.
File	Merupakan static method untuk membuat, copy, delete, move dan membuka file, dan membantu ketika membuat objek FileStream.
FileInfo	Merupakan non-static method untuk membuat, copy, delete, move dan membuka file, dan membantu ketika membuat objek FileStream
FileLoadException	Exception ketika file tidak bisa diload. Biasanya ada kerusakan pada isi file.

FileNotFoundException	Exception ketika file tidak bisa dibuka. Biasanya karena file tidak ada dan ada kerusakan pada isi media penyimpanan sehingga file tidak bisa dibuka.
FileStream	Class ini berguna untuk membuka dan menulis file dalam dua mode yaitu sinkronus atau asinkronus.
FileSystemEventArgs	Menyediakan data untuk event direktori : Changed, Created, Deleted.
FileSystemInfo	Menyediakan class dasar(base) untuk objek FileInfo dan DirectoryInfo.
IODescriptionAttribute	Memberi deskripsi tentang event, extender atau property.
IOException	Exception ketika eror I/O muncul
Path	Class ini bekerja untuk hal-hal yang berhubungan dengan informasi file atau direktori. Implementasi Path akan diperlihatkan di bawah.
PathTooLongException	Exception ini muncul ketika nama path atau nama file terlalu panjang dari panjang maximum system
Stream	Stream adalah abstract base class dari semua stream. Stream adalah abstraksi dari sekuen byte, seperti file, perangkat input/output, pipa komunikasi inter-proses, atau soket TCP/IP. Klas stream dan turunannya dapat memberikan tampilan secara generik dari tipe input dan output diatas, mengisolasi programmer dari detail sistem operasi dan beberapa perangkat.
StreamReader	Mengimplementasikan TextReader untuk membaca karakter-karakter dari byte stream.
StreamWriter	Mengimplementasikan TextWriter untuk menulis karakter-karakter ke dalam bentuk byte stream.
StringReader	Mengimplementasikan TextReader untuk membaca string.
StringWriter	Menulis kedalam bentuk string.
TextReader	Membaca karakter.
TextWriter	Menuliskan karakter.

13.2 Implementasi System.IO

13.2.1 Implementasi Class Path

Implementasi Class Path jika menggunakan windows form langkah-langkahnya adalah :
Buat satu windows application project baru.
drag 1 textbox dan 1 button lalu drop di form.
ketikan di dalam method button seperti ini :

```
private void button1_Click(object sender, System.EventArgs e)
{
    textBox1.Text = Path.GetFullPath("Form1.cs");
}
```

klik run, lalu tekan tombol.

13.2.2 Implementasi beberapa class System.IO

Implementasi beberapa class System.IO akan ditampilkan menggunakan console dan windows form.

Contoh menggunakan console adalah seperti ini :

```
string fileName = Path.GetTempFileName();
FileInfo fileInfo = new FileInfo(fileName);
Console.WriteLine("File '{0}' created of size {1} bytes",
    fileName, fileInfo.Length);

// Append some text to the file.
StreamWriter s = fileInfo.AppendText();
s.WriteLine("The text in the file");
s.Close();

fileInfo.Refresh();
Console.WriteLine("File '{0}' now has size {1} bytes",
    fileName, fileInfo.Length);

// Read the text file
StreamReader r = fileInfo.OpenText();
string textLine;
while ((textLine = r.ReadLine()) != null) {
    Console.WriteLine(textLine);
}
r.Close();
```

Jika menggunakan windows form langkah-langkahnya adalah :
Buat satu windows application project baru.
drag 1 button,1 textbox dan 1 richtextbox lalu drop di form.
ketikan di dalam method button seperti ini :

```
private void button1_Click(object sender, System.EventArgs e)
{
    string fileName = "C:\\COBA.TXT";
    FileInfo fileInfo = new FileInfo(fileName);
```

```
// menambahkan text ke file
StreamWriter s = fileInfo.AppendText();
s.WriteLine("The text in the file");
s.Close();

fileInfo.Refresh();

string textLine;
// membaca file text
StreamReader r = fileInfo.OpenText();
while ((textLine = r.ReadLine()) != null)
{
    textLine2 += textLine + "\n";
}

richTextBox1.Text = textLine2;

r.Close();
}
```

tambahkan "string textLine2" seperti ini :

```
public class Form1 : System.Windows.Forms.Form
{
    string textLine2;
    ...
}
```

klik run, lalu tekan tombol.

14. Delegate dan Event Handler

Risman Adnan

14.1. Sekilas tentang Delegate

Delegate adalah salah satu type dalam CTS (Common Type System) yang merupakan reference types. Delegate membungkus (encapsulate) suatu method (static maupun instance) dengan signature yang spesifik. Delegate hampir serupa dengan interface, karena keduanya menyatakan kontrak kerja antara caller dan implementer. Perbedaannya adalah interface tercipta pada saat kompilasi sedangkan delegate pada runtime. Penggunaan delegate yang paling umum adalah untuk mengimplementasikan fungsi callback dalam program dan event handling yang bersifat asynchronous. Karena terbentuk saat runtime, delegate juga dapat digunakan untuk membuat method (static maupun instance) yang tidak akan diketahui sampai saat runtime.

Dalam bab ini kita akan membahas delegate dari sudut pandang developer. Mula-mula kita akan membahas masalah apa yang dapat diselesaikan dengan adanya type delegate, selanjutnya kita akan melihat lebih jauh mengenai delegate dan perbandingannya dengan interface.

14.2. Delegate untuk Callback

Dibagian ini kita akan melihat contoh membuat dan menggunakan delegate. Misalkan kita punya class database manager yang berisi informasi mengenai semua koneksi aktif ke database. Class tersebut memiliki method untuk memberi enumerasi terhadap semua koneksi. Asumsikan bahwa class database manager ini berada di komputer yang terpisah, sehingga kita perlu membuat method yang asynchronous agar client dari class ini dapat melakukan callback. Perlu dicatat bahwa dalam aplikasi yang sebenarnya anda perlu membuat aplikasi ini sebagai aplikasi multithreading agar benar-benar bersifat asynchronous. Tapi untuk menyederhanakan permasalahan, kita bisa mengabaikan aspek multithreading untuk saat ini.

Mari kita lihat bagaimana cara mendefinisikan delegate dalam class yang memberikan callback.

```
class DBManager
{
    static koneksi DB[] koneksi Aktif;
    public delegate void EnumKoneksi Callback(koneksi DB koneksi);
    public static void EnumKoneksi (EnumKoneksi Callback callback)
    {
        foreach (koneksi DB koneksi in koneksi Aktif)
        {
            callback(koneksi);
        }
    }
}
```

Ada dua langkah untuk membuat suatu delegate sebagai callback dalam satu class atau server. Pertama, definisikan delegate yang sebenarnya, dalam hal ini EnumKoneksiCallback, yang akan menjadi signature dari method callback. Sintaks untuk mendefinisikan delegate adalah:

```
<Access Modifi er> delegate <type nilai balik> Nama method  
([Parameter])
```

Langkah kedua adalah definisikan suatu method yang salah satu parameter inputnya adalah delegate. Dalam contoh ini, method EnumKoneksi dengan parameternya instance dari delegate EnumKoneksiCallback.

Selanjutnya, user dari class ini (klien) hanya perlu mendefinisikan satu method yang memiliki signature yang sama dengan delegate, menggunakan operator new untuk membuat instance dari delegate, dan memasukkan nama instance tersebut ke method.

Berikut ini adalah contohnya:

```
public static void PrintKoneksi (koneksi DB koneksi )  
{  
}  
DBManager. EnumKoneksi Cal I back pri ntKoneksi  
= new DBManager. EnumKoneksi Cal I back(Pri ntKoneksi );
```

Pada akhirnya, klien memanggil method yang diinginkan dengan parameter berupa instance dari delegate.

Semua penjelasan di atas adalah sintaks utama dari delegate. Sekarang, mari kita lihat bentuk keseluruhan dari aplikasi ini.

```
usi ng System;  
usi ng System. Col I ecti ons;  
cl ass koneksi DB  
{  
    protected static int NmrKoneksi Beri kut = 1;  
    protected string namaKoneksi ;  
    public string NamaKoneksi  
    {  
        get  
        {  
            return namaKoneksi ;  
        }  
    }  
    public koneksi DB()  
    {  
        namaKoneksi = "Koneksi Database " +  
            koneksi DB. NmrKoneksi Beri kut++;  
    }  
}  
cl ass DBManager  
{  
    protected ArrayLi st koneksi Akti f;  
    public DBManager()  
    {  
        koneksi Akti f = new ArrayLi st();  
        for (int i = 1; i < 6; i++)
```

```

        {
            koneksi Akti f. Add(new koneksi DB());
        }
    }
    public delegate void EnumKoneksi Cal I back(
        koneksi DB koneksi );

    public void EnumKoneksi (EnumKoneksi Cal I back cal I back)
    {
        foreach(koneksi DB koneksi i n koneksi Akti f)
        {
            cal I back(koneksi );
        }
    }
};

class InstanceDel egate
{
    public static void Pri ntKoneksi (koneksi DB koneksi )
    {
        Consol e. Wri teLi ne(
            "[I nstanceDel egate. Pri ntKoneksi ]
            {0}", koneksi . NamaKoneksi );
    }
    public static void Main()
    {
        DBManager dbManager = new DBManager();
        Consol e. Wri teLi ne(" [Utama] I nstanti ate "
            + "method del egate");
        DBManager. EnumKoneksi Cal I back pri ntKoneksi =
        new DBManager. EnumKoneksi Cal I back(Pri ntKoneksi );
        Consol e. Wri teLi ne("[Mai n] Memanggil EnumKoneksi "
            + "- masukan del egate");
        dbManager. EnumKoneksi (pri ntKoneksi );
        Consol e. ReadLi ne();
    }
};

```

Hasil dari kompilasi program di atas ditunjukkan pada gambar berikut:

```

D:\INDC\OTAK\Bab14\DelegateCallback\bin\Debug\DelegateCallback.exe
[Utama] Instantiate method delegate
[Main] Memanggil EnumKoneksi - masukan delegate
[InstanceDelegate.PrintKoneksi] Koneksi Database 1
[InstanceDelegate.PrintKoneksi] Koneksi Database 2
[InstanceDelegate.PrintKoneksi] Koneksi Database 3
[InstanceDelegate.PrintKoneksi] Koneksi Database 4
[InstanceDelegate.PrintKoneksi] Koneksi Database 5

```

Gambar 14-1. Sangat mudah mengimplementasikan callback dengan delegate

Anda telah melihat betapa mudahnya mendefinisikan method callback menggunakan delegate. Method callback memungkinkan anda untuk melewati satu pointer fungsi

ke fungsi lainnya yang selanjutnya akan menghasilkan callback. Callback digunakan untuk banyak keperluan, diantaranya yang paling sering adalah untuk:

14.2.1 Proses Asynchronous

Skenario asynchronous dapat digambarkan sebagai berikut: Kode klien membuat pemanggilan terhadap suatu method melalui method callback. Method yang dipanggil melakukan tugasnya dan kembali memanggil fungsi callback setelah tugasnya selesai. Ini akan memberi keuntungan karena klien dapat melanjutkan pekerjaannya tanpa terhenti oleh pemanggilan fungsi yang synchronous terlalu lama. Dalam kehidupan sehari-hari, proses asynchronous ini sering terjadi. Misalnya anda membutuhkan nomor telpon teman anda Agus Kurniawan. Satu-satunya nomornya yang anda bisa hubungi adalah nomor dari teman anda Kunarto. Anda bisa menelpon Kunarto untuk meminta bantuan, Kunarto akan mencari nomor telpon Agus Kurniawan dan menelpon balik ke anda jika nomor tersebut telah diketahuinya.

14.2.2 Memasukkan Kode Tambahan ke Kode Suatu Class

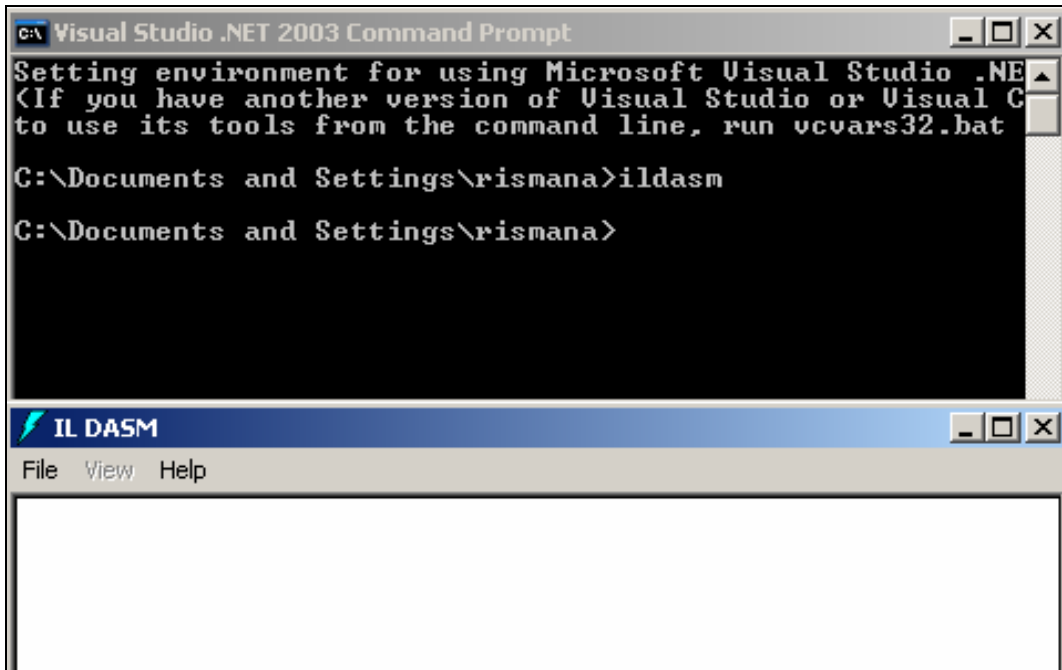
Method callback juga sering digunakan ketika suatu class mengizinkan kliennya untuk menentukan suatu *method* yang akan dipanggil dalam melakukan proses tertentu. Contoh yang sering anda lihat adalah pada aplikasi Windows yang menggunakan *ListBox*. Dengan menggunakan *class ListBox* dalam Windows, anda dapat menentukan bahwa item-item dalam *ListBox* dapat diurutkan secara *ascending* maupun *descending*. Disamping beberapa pilihan pengurutan biasa, *class ListBox* tetap merupakan *class* yang generik. Oleh sebab itu, *ListBox* juga memungkinkan anda untuk membuat suatu fungsi *callback* untuk pengurutan. Sedemikian hingga saat *ListBox* mengurutkan item-itemnya, *ListBox* akan memanggil fungsi *callback* dan anda bisa melakukan pengurutan tambahan (selain *ascending* dan *descending*) yang diperlukan di dalam kode program anda.

Selanjutnya kita akan melihat lebih jauh apa yang ada dalam delegate dengan menggunakan *Microsoft Intermediate Language* (MSIL). Anda dapat menggunakan *Visual Studio .NET Command Prompt* untuk memanggil program ILDASM (Program untuk melakukan disassembly terhadap file assembly, dll atau exe).

14.3 Delegate Adalah Class

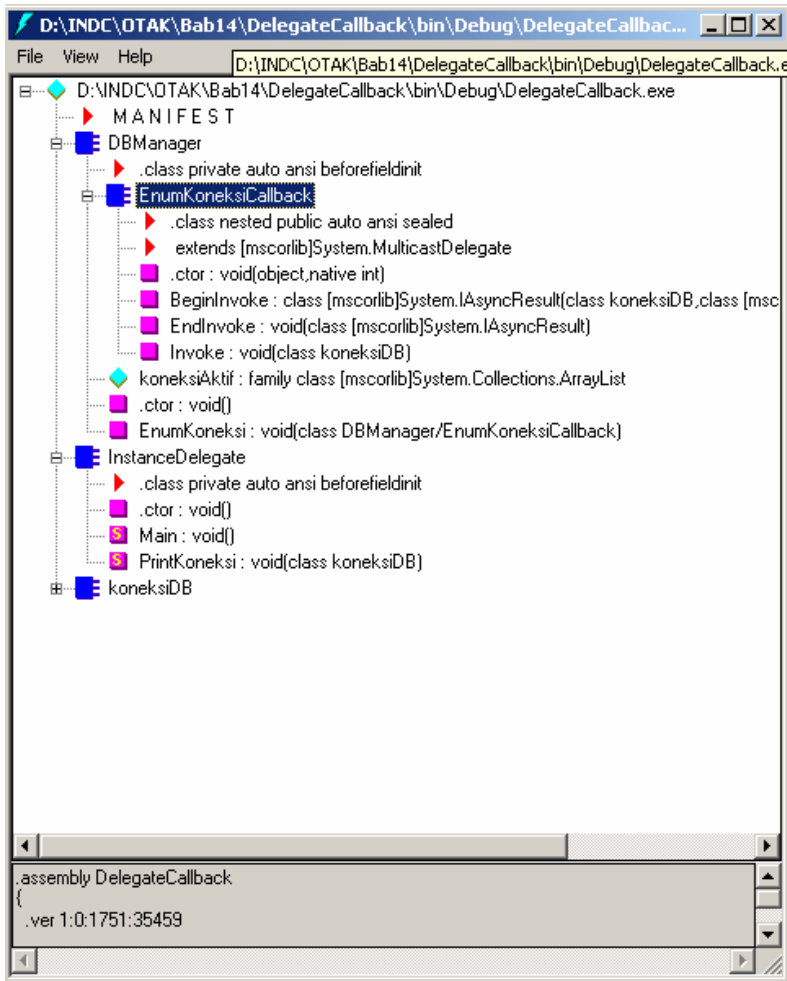
Jika anda pernah bekerja dengan C++, mungkin anda bisa menebak bahwa *delegate* menyerupai fungsi *pointer* C++. Beberapa buku menjelaskan *delegate* sebagai fungsi pointer yang telah bersifat *type-safe*. Akan tetapi, seperti yang telah anda lihat sebelumnya, sekali *delegate* didefinisikan, kita dapat membuat *instance* dari *delegate* seperti halnya pada *class* dengan menggunakan operator *new*. Sebenarnya, *delegate* dapat didefinisikan sebagai ekivalen dari **Functor** dalam C++. *Functor* atau fungsi obyek adalah *class* C++ yang meng-*overload* operator tanda kurung sehingga hasilnya terlihat seakan-akan sebagai suatu fungsi walaupun sebenarnya adalah *class polymorphic*.

Untuk membuka MSIL dari program anda sebelumnya, gunakan *Visual Studio .NET Command Prompt* dan panggil program ILDASM. Buka file assembly **DelegateCallback.exe** yang ada di contoh sebelumnya dengan ILDASM.



Gambar 14-2. Cara memanggil program ILDASM melalui command prompt.

Dari kode MSIL yang anda peroleh terlihat bahwa delegate cenderung lebih mendekati Functor dibanding fungsi pointer sederhana. Hal ini ditunjukkan pada gambar 14.3, EnumKoneksiCallback adalah *class polymorphic*.



Gambar 14-3. *Delegate* pada dasarnya adalah *class*

Seperti yang anda lihat, *delegate* yang ada dalam contoh sebelumnya (*EnumKoneksiCallback*) adalah *class* yang diturunkan dari *System.MulticastDelegate* dan terdiri dari anggota-anggota berikut:

- Konstruktor *EnumKoneksiCallback*
- *Invoke*
- *BeginInvoke*
- *EndInvoke*

Anda bisa meneliti lebih jauh isi dari anggota-anggota *class* tersebut di atas untuk memperoleh gambaran mengenai *delegate*.

14.4 Mendefinisikan Delegate Sebagai Static Member

Delegate juga dapat dibentuk dari anggota *class* yang bersifat *static*. *Delegate* pada contoh di bawah ini didefinisikan sebagai anggota *static* dari *class* `printKoneksi`. Klien tidak perlu membuat *instance* dari *delegate* dalam *method* `Main`.

```

using System;
using System.Collections;
class koneksi DB
{
    protected static int NmrKoneksi Berikut = 1;
    protected string namaKoneksi;
    public string NamaKoneksi
    {
        get
        {
            return namaKoneksi;
        }
    }
    public koneksi DB()
    {
        namaKoneksi = "Koneksi Database "
            + koneksi DB. NmrKoneksi Berikut++;
    }
}

class DBManager
{
    protected ArrayList koneksi Aktif;
    public DBManager()
    {
        koneksi Aktif = new ArrayList();
        for (int i = 1; i < 6; i++)
        {
            koneksi Aktif. Add(new koneksi DB());
        }
    }
    public delegate void EnumKoneksi Cal I back(
        koneksi DB koneksi);

    public void EnumKoneksi (EnumKoneksi Cal I back cal I back)
    {
        foreach(koneksi DB koneksi i n koneksi Aktif)
        {
            cal I back(koneksi);
        }
    }
};

class Stati cDel egate
{
    static DBManager. EnumKoneksi Cal I back pri ntKoneksi
        = new DBManager. EnumKoneksi Cal I back(Pri ntKoneksi);
    public static void Pri ntKoneksi (koneksi DB koneksi)
    {
        Console. Wri teLi ne(
            "[Stati cDel egate. Pri ntKoneksi ] {0}",
            koneksi . NamaKoneksi );
    }
    public static void Mai n()
    {
        DBManager dbManager = new DBManager();
        Console. Wri teLi ne("[Mai n] Memangi l EnumKoneksi ")

```

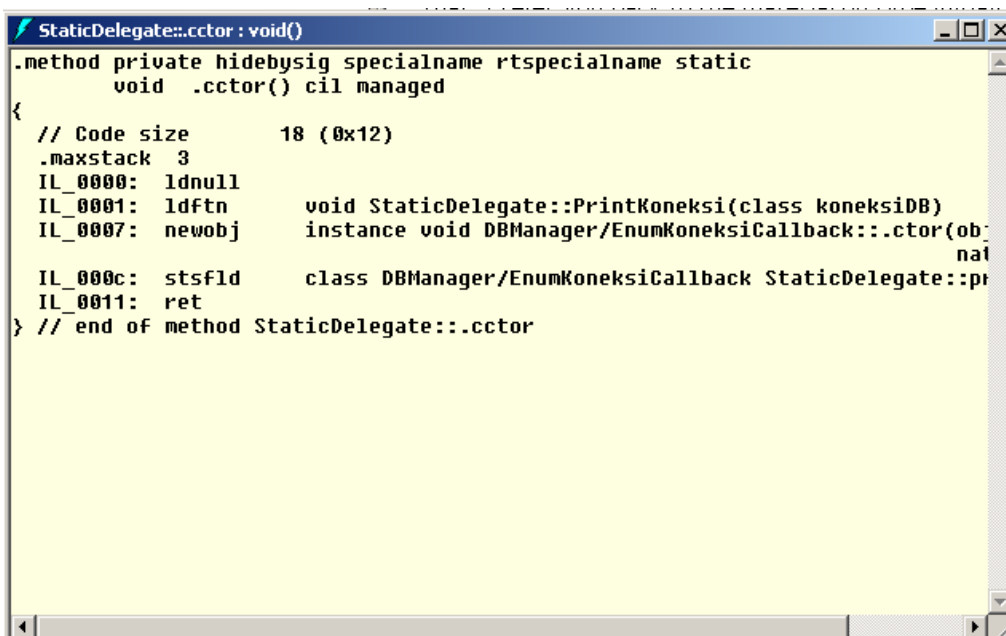
```

        + "- masukkan delegate");
        dbManager.EnumKoneksi (pri ntKoneksi );
        Console.ReadLine();
    }
};

```

Ketika anda menjalankan program di atas, hasil yang diperoleh sama dengan yang sebelumnya. Akan tetapi, obyek *delegate* sekarang bersifat *static*. Bagaimana hal ini terjadi? Anda mungkin akan menebak bahwa jika *delegate* dideklarasikan sebagai *static*, seharusnya tetap ada *instance* dari *delegate* tersebut, tapi dimana?

Dalam Common Language Infrastructure (CLR), suatu *type static* dapat berisi *method* yang dikenal sebagai *type initializer* yang bisa menginisialisasi dirinya sendiri. Aturan utama dari *method* ini adalah, harus *static*, tidak memiliki parameter, tidak ada nilai balik dan dinamakan *.cctor*. Anda bisa melihat *method* ini dari MSIL seperti pada gambar berikut.



Gambar 14-4. *Method .cctor* adalah satu *type initializer* yang digunakan oleh *type static* untuk menginisialisasi dirinya.

14.5 Property dan Delegate

Saat mendefinisikan *delegate*, anda harus memperhatikan kapan *delegate* tersebut dibuat. Katakanlah, proses pembuatan *delegate* tersebut memakan waktu dalam eksekusi program anda. Dalam situasi dimana anda tahu bahwa klien tidak akan memanggil *method callback*, kita dapat menunda pembuatan *delegate* tersebut sampai benar-benar diperlukan dengan menggunakan *property*. Contoh berikut ini menggambarkan penggunaan *property read-only* untuk membuat *instance* dari *delegate*.

```
using System;
using System.Collections;
class koneksiDB
{
    protected static int NmrKoneksiBerikut = 1;
    protected string namaKoneksi;
    public string NamaKoneksi
    {
        get
        {
            return namaKoneksi;
        }
    }

    public koneksiDB()
    {
        namaKoneksi = "Koneksi Database "
            + koneksiDB.NmrKoneksiBerikut++;
    }
}

class DBManager
{
    protected ArrayList koneksiAktif;
    public DBManager()
    {
        koneksiAktif = new ArrayList();
        for (int i = 1; i < 6; i++)
        {
            koneksiAktif.Add(new koneksiDB());
        }
    }

    public delegate void EnumKoneksiCallback(
        koneksiDB koneksi);
    public void EnumKoneksi(EnumKoneksiCallback callback)
    {
        foreach(koneksiDB koneksi in koneksiAktif)
        {
            callback(koneksi);
        }
    }
};

class DelegateProperty
{
    void printKoneksi(koneksiDB koneksi)
    {
        Console.WriteLine("[DelegateProperty.PrintKoneksi]
            {0}", koneksi.NamaKoneksi);
    }

    public DBManager.EnumKoneksiCallback PrintKoneksi
    {
        get
        {
            return new
                DBManager.EnumKoneksiCallback(printKoneksi);
        }
    }

    public static void Main()
    {

```

```

        DelegateProperty app = new DelegateProperty();
        DBManager dbManager = new DBManager();
        Console.WriteLine("[Main] Memanggil EnumKoneksi "
            + "- masukkan delegate");
        dbManager.EnumKoneksi (app. PrintKoneksi );
        Console.ReadLine();
    }
};

```

Seperti yang anda lihat, yang perlu dilakukan adalah:

- Definisikan *method* yang akan berlaku sebagai *delegate receiver* dengan *access modifier* yang membatasi akses eksternal.
- Definisikan *property get* yang memberikan suatu *instance delegate* untuk *method receiver*. Ketika diinvoke, *property* ini menghasilkan obyek *delegate* menggunakan *method receiver*.
- Panggil *method* yang menggunakan obyek *delegate* sebagai parameter.

Pelajaran apa yang dapat anda peroleh dari bagian ini? Ternyata *property* dapat membantu anda untuk membuat perantara yang lebih *elegant* untuk membuat *instance* dari obyek *delegate*.

14.6 Multicast Delegate

Kita dapat membuat Multicast Delegate dengan cara mengkombinasikan beberapa delegate menjadi satu delegate tunggal. Awalnya konsep ini akan menyulitkan buat anda, tapi anda akan sangat berterima kasih kepada tim pembuat CLI disaat anda menghadapi kasus dimana dibutuhkan Multicast Delegate. Mari kita lihat contoh dimana Multicast Delegate sangat berguna. Pada contoh pertama, kita memiliki sistem inventory dan class InventoryManager yang bertugas untuk melakukan pendataan bahan material pada satu lokasi warehouse. Pendataan ini menghasilkan jumlah bahan yang tersedia bergantung pada nomor SKU (Stock Keeping Unit). Untuk menyederhanakan permasalahan, anda bisa menggunakan bilangan acak untuk menghasilkan jumlah bahan. Pada tiap iterasi dalam pendataan bahan, class InventoryManager akan memanggil method callback untuk mendeteksi apakah jumlah bahan yang tersedia lebih kecil dari nilai minimum yaitu 50. Satu delegate cukup untuk membuat program ini, tetapi bagaimana jika anda harus mencatat kejadian (event) disaat anda menemukan jumlah barang yang tersedia lebih kecil dari nilai minimum dan langsung mengirim email laporan ke Purchasing Manager tentang kejadian ini? Tentu saja anda perlu membuat satu delegate gabungan dari beberapa delegate, perhatikan kode program berikut:

```

using System;
using System.Threading;

class Bahan
{
    public Bahan(string sku)
    {
        this.Sku = sku;

        Random r = new Random(DateTime.Now.Millisecond);
        double d = r.NextDouble() * 100;

        this.Tersedia = (int)d;
    }
}

```

```

        protected string sku;
        public string Sku
        {
            get { return this.sku; }
            set { this.sku = value; }
        }

        protected int tersedia;
        public int Tersedia
        {
            get { return this.tersedia; }
            set { this.tersedia = value; }
        }
    };

    class InventoryManager
    {
        protected const int MIN_TERSEDIA = 50;
        public Bahan[] bahan;
        public InventoryManager()
        {
            Console.WriteLine("[InventoryManager. InventoryManager]"
+ " Menambah Bahan...");
            bahan = new Bahan[5];
            for (int i = 0; i < 5; i++)
            {
                Bahan bhn = new Bahan("Bahan " + (i + 1));
                Thread.Sleep(10); // Randomizer is seeded by
time.
                bahan[i] = bhn;
                Console.WriteLine("\tPart '{0}' on-hand = {1}",
                    bhn.Sku, bhn.Tersedia);
            }
        }

        public delegate void OutOfStockExceptionMethod(Bahan
bahan);
        public void ProcessInventory(OutOfStockExceptionMethod
exception)
        {
            Console.WriteLine("\n[InventoryManager. ProcessInventory]"
+ " Melakukan Proses Inventory...");
            foreach (Bahan bhn in bahan)
            {
                if (bhn.Tersedia < MIN_TERSEDIA)
                {
                    Console.WriteLine("\n\t{0} ({1} units) =
" +
                    "di bawah angka minimum tersedia
{2}",
                    bhn.Sku,
                    bhn.Tersedia,
                    MIN_TERSEDIA);
                    exception(bhn);
                }
            }
        }
    };

    class ComposeDelegate
    {

```

```

public static void LogEvent(Bahan bahan)
{
    Console.WriteLine("\t[Composi teDel egate. LogEvent] " +
        "mencatat event...");
}

public static void EmailPurchasingMgr(Bahan bahan)
{
    Console.WriteLine("\t[Composi teDel egate" +
        ". EmailPurchasingMgr] Kirim email ke
manager...");
}

public static void Main()
{
    InventoryManager mgr = new InventoryManager();
    InventoryManager. OutOfStockExcepti onMethod
        LogEventCall back = new
InventoryManager. OutOfStockExcepti onMethod(LogEvent);
    InventoryManager. OutOfStockExcepti onMethod
        EmailPurchasingMgrCall back = new
InventoryManager. OutOfStockExcepti onMethod(
        EmailPurchasingMgr);
    InventoryManager. OutOfStockExcepti onMethod
        OnHandExcepti onEventsCall back =
        EmailPurchasingMgrCall back + LogEventCall back;
    mgr. ProcessInventory(OnHandExcepti onEventsCall back);
    Console. ReadLi ne();
}
};

```

Jika anda menjalankan program ini, hasilnya ditunjukkan pada gambar 14.5. Pelajaran apa yang anda peroleh? Ternyata dalam C# kita bisa secara dinamik menentukan *method-method* yang berhubungan dengan suatu *method callback*, menggabungkan *method-method* tersebut ke dalam satu *delegate* dengan operator *plus (+)*. *Runtime* akan secara otomatis akan menjamin bahwa *method-method* tersebut dipanggil secara berurutan. Lebih jauh lagi, anda dapat mengeluarkan *delegate* dari gabungannya dengan menggunakan operator *minus (-)*.

```

D:\INDC\OTAK\Bab14\CompositeDelegate\bin\Debug\CompositeDelegate.exe
[InventoryManager.InventoryManager] Menambah Bahan...
Part 'Bahan 1' on-hand = 65
Part 'Bahan 2' on-hand = 87
Part 'Bahan 3' on-hand = 10
Part 'Bahan 4' on-hand = 32
Part 'Bahan 5' on-hand = 55

[InventoryManager.ProcessInventory] Melakukan Proses Inventory...

Bahan 3 (10 units) = dibawah angka minimum tersedia 50
[CompositeDelegate.EmailPurchasingMgr] Kirim email ke manager...
[CompositeDelegate.LogEvent] mencatat event...

Bahan 4 (32 units) = dibawah angka minimum tersedia 50
[CompositeDelegate.EmailPurchasingMgr] Kirim email ke manager...
[CompositeDelegate.LogEvent] mencatat event...
    
```

Gambar 14.5. *Multicast delegate* memungkinkan anda untuk menggabungkan beberapa obyek *delegate* menjadi satu *delegate* tunggal.

14.7. Mendefinisikan Event Dari Multicast Delegate

Dalam pengembangan aplikasi Windows, anda akan sering menemukan kasus dimana *asynchronous-event processing* sangat diperlukan. Beberapa *event* ini sangat generik, misalnya aplikasi harus mengirim *message* ke MSMQ disaat user berinteraksi dengan GUI. Contoh lainnya yang lebih spesifik, ketika *invoice* dicetak, semua tiket pengambilan barang harus di update. Dalam .NET, skenario *asynchronous-event processing* dapat diimplementasikan dengan *multicast delegate* dan *keyword event* dalam C#.

Model *multicast delegate* berhubungan erat dengan *Observer Pattern*, atau *Publish/Subscribe Pattern*, dimana suatu class mem-publish suatu event yang dapat dimunculkan olehnya, dan class lainnya dapat men-subscribe ke event tersebut. Disaat event terjadi, *runtime engine* akan memberitahu *subscriber* bahwa event itu terjadi. *Method* yang dipanggil akibat terjadinya suatu event dapat didefinisikan oleh delegate. Walaupun penggunaan *delegate* dalam kasus ini terkesan mudah, perlu diperhatikan hal-hal berikut:

Delegate harus didefinisikan dengan dua *argument* (parameter input) *Argument-argument* selalu merepresentasikan dua obyek, yaitu obyek yang menyebabkan event (*publisher*) dan obyek informasi event. Obyek kedua harus diturunkan dari class *EventArgs* yang ada dalam .NET Framework.

Untuk melihat bagaimana implementasi event akan memberikan design sistem yang lebih baik, mari kita lanjutkan contoh **InventoryManager** sebelumnya. Kali ini kita ingin memiliki kemampuan untuk memonitor perubahan-perubahan pada tingkatan *inventory*. Salah satu yang termasuk dalam design adalah satu class yang dinamakan **InventoryManager** yang selalu digunakan untuk meng-update *inventory*. Class **InventoryManager** ini akan mem-publish suatu event jika yang akan terjadi setiap saat *inventory* berubah, misalnya dengan adanya penerimaan bahan baku, penjualan dan *stock opname*. Dengan demikian semua class yang perlu selalu ter-update harus

melakukan *subscribe* ke *event*. Berikut ini adalah kode program dalam C# yang menggunakan *delegate* dan *event*.

```
using System;
class InventoryChangeEventArgs : EventArgs
{
    public InventoryChangeEventArgs(string sku, int change)
    {
        this.sku = sku;
        this.change = change;
    }
    string sku;
    public string Sku
    {
        get { return sku; }
    }
    int change;
    public int Change
    {
        get { return change; }
    }
};

class InventoryManager // Publisher
{
    public delegate void InventoryChangeEventHandler(
        object source, InventoryChangeEventArgs e);
    public event InventoryChangeEventHandler
        OnInventoryChangeHandler;

    public void UpdateInventory(string sku, int change)
    {
        if (0 == change)
            return; // Tidak ada update tanpa perubahan

        // Kode untuk update database letakkan di sini

        InventoryChangeEventArgs e =
            new InventoryChangeEventArgs(sku, change);

        if (OnInventoryChangeHandler != null)
        {
            Console.WriteLine("[InventoryManager] +
                ".UpdateInventory] Membuat event ke semua
" +
                "Subscriber... \n");
            OnInventoryChangeHandler(this, e);
        }
    }
};

class InventoryWatcher // Subscriber
{
    public InventoryWatcher(InventoryManager inventoryManager)
    {
        Console.WriteLine("[InventoryWatcher] +
            ".InventoryWatcher] Subscribe ke " +
            "InventoryChange event\n");
        this.inventoryManager = inventoryManager;
        inventoryManager.OnInventoryChangeHandler += new
            InventoryManager.InventoryChangeEventHandler(
            OnInventoryChange);
    }
}
```

```
void OnInventoryChange(object source,
    InventoryChangeEventArgs e)
{
    int change = e.Change;

    Console.WriteLine("[InventoryManager.OnInventoryChange]" +
        "\n\tBahan '{0}' {1} sebanyak {2} unit\n",
        e.Sku,
        change > 0 ? "bertambah" : "berkurang",
        Math.Abs(e.Change));
}
InventoryManager inventoryManager;
}

class DelegateEvents
{
    public static void Main()
    {
        InventoryManager inventoryManager =
            new InventoryManager();

        Console.WriteLine("[DelegateEvents.Main]" +
            "Membuat instance Obyek Subscriber\n");
        InventoryWatcher inventoryWatch =
            new InventoryWatcher(inventoryManager);

        inventoryManager.UpdateInventory("111 006 116", -2);
        inventoryManager.UpdateInventory("111 005 383", 5);

        Console.ReadLine();
    }
};
```

Kode di atas adalah contoh yang paling baik untuk mempelajari delegate dan event. Anda perlu melihat kode tersebut baris per baris untuk mendapatkan pemahaman yang lengkap tentang delegate dan event.

Untuk menghasilkan design yang baik, anda perlu mengenal beberapa pattern yang umum diketahui dalam pemrograman berorientasi obyek (OOP). Pembahasan mengenai *design pattern* akan kita tunda untuk buku INDC selanjutnya*. Topik *design pattern* adalah kelanjutan dari topik pemrograman berorientasi obyek bagi developer C# yang telah mempelajari banyak aspek OOP.

15. Studi Kasus - Penerapan OOP Dengan C#

Agus Kurniawan

15.1 Pendahuluan

Setelah kita mempelajari beberapa konsep OOP mulai dari Class hingga inheritance maka saatnya sekarang kita menerapkan konsep OOP dalam project yang realitas di lapangan. Studi kasus yang akan dibuat dalam bab ini adalah membuat aplikasi untuk perhitungan gaji disuatu perusahaan.

15.2 Membangun Aplikasi Perhitungan Gaji

Kita telah mengetahui bahwa dalam suatu perusahaan terutama dibagian HRD sering disibukkan dengan perhitungan gaji semua karyawan di perusahaan tersebut. Di bab ini, kita akan membuat aplikasi perhitungan gaji dengan menerapkan konsep OOP pada C#.

15.3 Design Aplikasi

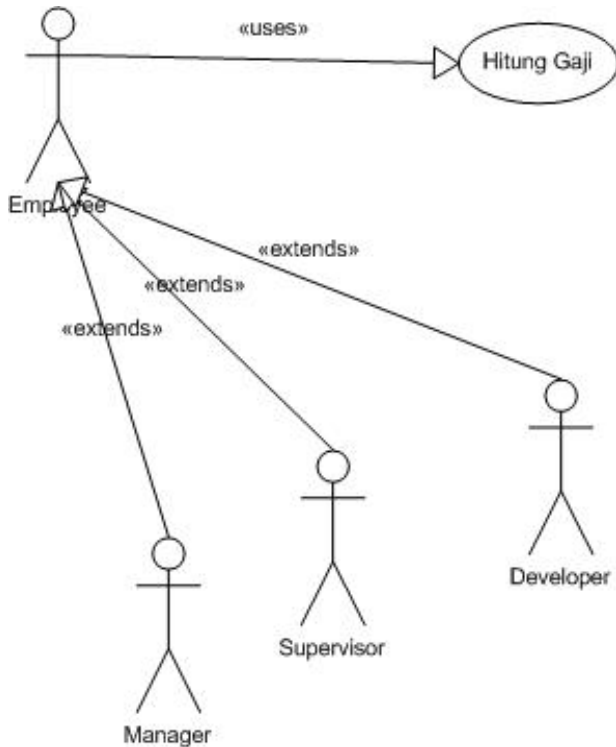
Ada 2 solusi yang dapat kita dilakukan untuk membuat aplikasi perhitungan gaji karyawan suatu perusahaan yaitu:

- Dengan menerapkan konsep inheritance
- Dengan menerapkan konsep interface

Masing-masin solusi ini akan diimplementasikan kedalam bab ini.

15.4 Diagram Use Cases

Diagram use cases untuk scenario ini dapat dilihat pada gambar dibawah ini:



Gambar 15-1. Use Case diagram

Kalau diperhatikan pada gambar diatas maka kita dapatkan 3 actor yang terlibat dalam project ini. Actor tersebut antara lain:

- Manager
- Supervisor
- Developer

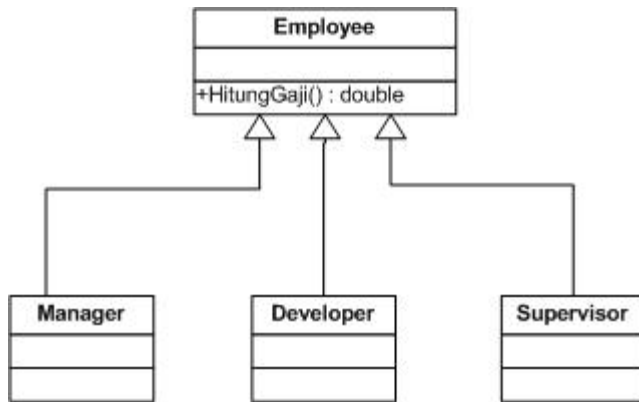
Ketiga actor ini juga merupakan inheritance dari actor Employee. Didalam actor Employee terdapat juga satu case yaitu Hitung gaji.

15.5 Diagram Class

Di sub bab ini kita akan membuat 2 diagram class sesuai dengan solusi yang dideklarasikan pada 15.3 yaitu dengan memanfaatkan inheritance dan interface.

15.5.1 Menggunakan Inheritance

Kalau solusi perhitungan gaji dengan menggunakan inheritance maka kita dapat memanfaatkan reusability suatu objek. Dibawah ini adalah diagram class untuk menghitung gaji dengan konsep inheritance.



Gambar 15-2. Perhitungan gaji dengan konsep inheritance

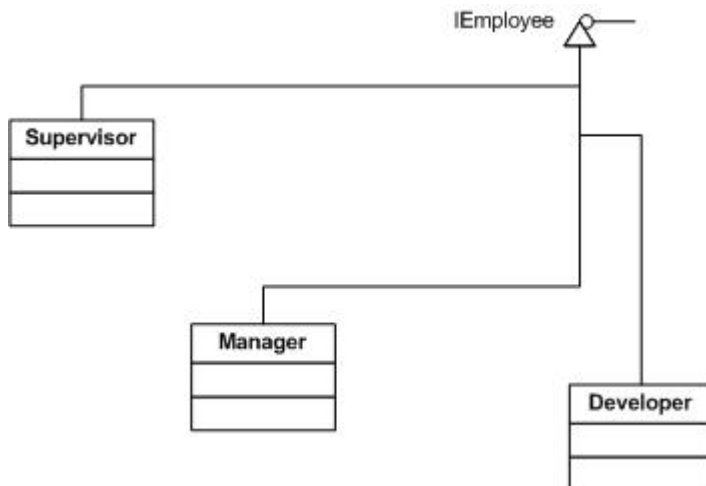
Pada gambar 15-2 terlihat ada 4 Class antara lain:

- Employee
- Manager
- Developer
- Supervisor

Kalau kita perhatikan lebih dalam maka Class Manager, Developer, dan Supervisor merupakan inheritance dari Class Employee.

15.5.2 Menggunakan Interface

Untuk solusi ke-2, kita dapat memanfaatkan interface untuk membungkus kekompleksitas suatu sistem. Oleh karena itu, objek yang mengaksesnya tidak perlu mengetahui objek apa yang diinstansiasi, cukup hanya mengetahui methodnya. Berikut ini adalah gambar diagram class untuk menghitung gaji dengan memanfaatkan konsep interface.



Gambar 15-3 Perhitungan gaji dengan konsep interface

Pada gambar diatas, masing-masing Class Supervisor, Manager, dan Developer merupakan implementasi dari interface IEmployee.

15.6 Implementasi Diagram Class

Langkah selanjutnya kita akan mengimplementasi diagram class kedalam code C#.

15.6.1 Menggunakan Konsep Inheritance

Kalau kita perhatikan pada gambar 15-2 maka ada 4 Class yang akan diimplementasikan kedalam code C#. Pertama adalah Class Employee yang merupakan Class parent (Class induk) karena Class ini akan dibuat inheritance untuk Class Manager, Developer, dan Supervisor.

Berikut ini implementasi Class Employee:

```
public class Employee
{
    public Employee()
    {
    }

    public virtual double HitungGaji()
    {
        return 2000.00;
    }
}
```

Pada Class Employee, method HitungGaji() merupakan virtual method supaya Class-Class yang merupakan inheritance dari Class ini dapat melakukan override. Untuk Class Employee, method HitungGaji() mengembalikan nilai 2000.00

Kemudian kita mengimplementasikan Class Manager yang inheritance dari Class Employee sebagai berikut:

```
public class Manager: Employee
{
    public Manager()
    {
    }

    public override double HitungGaji()
    {
        return 5*base.HitungGaji();
    }
}
```

Pada Class ini, method HitungGaji() dilakukan override dan dikalikan nilai 5 dari nilai gaji Class Employee

Dengan cara yang sama maka kita dapat mengimplementasikan juga untuk Class Developer dan Supervisor. Berikut ini code implementasinya:

```
public class Developer: Employee
{
    public Developer()
    {
    }

    public override double HitungGaji()
    {
```

```
        return 2*base.HitungGaji ();
    }
}

public class Supervisor: Employee
{
    public Supervisor()
    {
    }

    public override double HitungGaji ()
    {
        return 3*base.HitungGaji ();
    }
}
```

15.6.2 Menggunakan Konsep Interface

Kita dapat melihat gambar 15-3 yang merupakan penerapan Interface untuk menghitung gaji karyawan. Pada gambar tersebut terdapat interface yaitu IEmployee, dibawah ini implementasinya kedalam code:

```
public interface IEmployee
{
    double HitungGaji ();
}
```

Untuk implementasi interface IEmployee pada Class Manager, kita cukup implementasi method yang ada didalam interface IEmployee. Berikut contoh code untuk penerapannya:

```
public class Manager: IEmployee
{
    public Manager()
    {
    }

    public double HitungGaji ()
    {
        return 10000.00;
    }
}
```

Dengan cara yang sama, kita dapat implementasinya untuk Class Developer dan Supervisor sebagai berikut.

```
public class Supervisor: IEmployee
{
    public Supervisor()
    {
    }

    public double HitungGaji ()
    {
        return 7000.00;
    }
}
```

```
public class Developer: Employee
{
    public Developer()
    {
    }

    public double HitungGaji()
    {
        return 5000.00;
    }
}
```

15.7 Menjalankan Aplikasi

Ok, sub bab ini kita akan menjalankan Class-Class yang telah dibuat sesuai dengan solusi yang dipakai.

15.7.1 Konsep Inheritance

Pemakaian Class-Class untuk menghitung gaji dapat diletakan pada method main kemudian kita dapat menjalankan aplikasi ini.

Berikut ini penggunaan Class-Class tersebut:

```
class Class1
{
    static void Main(string[] args)
    {
        Employee pekerja = new Employee();
        Console.WriteLine("Gaji Employee : {0}",
            pekerja.HitungGaji());

        pekerja = new Manager();
        Console.WriteLine("Gaji Manager : {0}",
            pekerja.HitungGaji());

        pekerja = new Supervisor();
        Console.WriteLine("Gaji Supervisor : {0}",
            pekerja.HitungGaji());

        pekerja = new Developer();
        Console.WriteLine("Gaji Developer : {0}",
            pekerja.HitungGaji());

        Console.ReadLine();
    }
}
```


Keterangan code:

Mula-mula kita mendeklarasikan variabel pekerja yang bertipe Employee dan kita panggil method HitungGaji

```
Empl oyee pekerj a = new Empl oyee();  
Consol e. Wri teLi ne("Gaj i Empl oyee : {0}", pekerj a. Hi tungGaj i ());
```

Kemudian kita melakukan instansiasi objek pekerja dengan tipe Manager dan memanggil method HitungGaji

```
pekerj a = new Manager();  
Consol e. Wri teLi ne("Gaj i Manager : {0}", pekerj a. Hi tungGaj i ());
```

Dengan cara yang sama juga dapat dilakukan untuk tipe Class Supervisor dan Developer. Dibawah ini adalah hasil compile codenya:



```
D:\My Document\Community C# Indonesia\Kurikulum\Release\CSH101 - Pengenalan Bahasa C#\Cod...  
Gaji Employee : 2000  
Gaji Manager : 10000  
Gaji Supervisor : 6000  
Gaji Developer : 4000
```

Gambar 15-4. Hasil running aplikasi hitung gaji dengan konsep inheritance

15.7.2 Konsep Interface

Sedangkan pemakaian Class-Class untuk menghitung gaji dengan konsep interface dapat ditulis seperti code dibawah ini:

```
cl ass Cl ass1  
{  
    [STAThread]  
    stati c voi d Mai n(stri ng[] args)  
    {  
        I Empl oyee I Karyawan = nul l ;  
  
        I Karyawan = new Manager();  
        Consol e. Wri teLi ne("Gaj i Manager : {0}",  
            I Karyawan. Hi tungGaj i ());  
  
        I Karyawan = new Supervi sor();
```

```
Console.WriteLine("Gaji Supervisor : {0}",  
    I Karyawan. HitungGaji ());  
  
I Karyawan = new Developer();  
Console.WriteLine("Gaji Developer : {0}",  
    I Karyawan. HitungGaji ());  
  
Console.ReadLine();  
  
    }  
}
```

Keterangan code:

Pertama kali mendeklarasikan variabel interface IEmployee dan disetting dengan nilai null

```
I Employee I Karyawan = null ;
```

Selanjutnya kita melakukan instansiasi objek IEmployee dengan objek Class Manager dan memanggil method HitungGaji

```
I Karyawan = new Manager();  
Console.WriteLine("Gaji Manager : {0}", I Karyawan. HitungGaji ());
```

Dengan cara yang sama juga dapat dilakukan untuk tipe Class Supervisor dan Developer. Dibawah ini adalah hasil compile codenya:



```
D:\My Document\Community C# Indonesia\Kurikulum\Release\CSH101 - Pengenalan Bahasa C#\Cod...  
Gaji Manager : 10000  
Gaji Supervisor : 7000  
Gaji Developer : 5000
```

Gambar 15-5. Hasil running aplikasi hitung gaji dengan konsep interface

Daftar Pustaka

1. MSDN Library
2. Sintes Anthony, 2002, Teach Yourself Object Oriented Programming in 21Days, Sams, Indiana.
3. Cornell Gary and Horstmann Cay S, 1997, Core Java Edisi Indonesia bab 4-bab 5, Andi Yogyakarta.
4. Kanetkar's Yashavant, 2002, C# .Net Fundas, Tech Publications Pte Ltd, Singapore
5. Schildt Herbert, 2001, C# a Beginner's Guide, Osborne/McGraw-Hill, California.
6. W. Cooper James, 2002, Introduction To Design Patterns in C#, IBM T J Watson Research Center
7. Standard ECMA-334, December 2001, C# Language Specification. <http://www.ecma.ch>
8. Microsoft Corporation, "Microsoft Official Curcullum 2124C: Programming with C#", USA, 2002
9. Robinson, Simon, & friends, "Professional C# 2nd Edition", Wrox, Birmingham, UK, 2002
10. "Thinking in C#", Prentice Hall PTR, 2002

Lampiran

MSDN Connection

MSDN Connection adalah komunitas developer .NET di Indonesia. Komunitas ini bertujuan meningkatkan pemahaman dan pengalaman anda tentang .NET. Dengan bergabung bersama kami, anda otomatis mendapatkan update secara berkala mengenai perkembangan teknologi .NET langsung dari Microsoft Indonesia. Jika anda ingin bergabung dengan kami, caranya mudah. Hanya dengan mengklik url <http://www.msdnconnection.com/indonesia> dan masukkan data-data anda.

Komunitas C# Indonesia

Komunitas C# Indonesia merupakan bagian dari komunitas .NET Indonesia (INDC) yang berfokus kebidang teknologi .NET dengan menggunakan bahasa C#.

Komunitas C# Indonesia dapat dijumpai di url www.csharpindonesia.net dan anda dapat bergabung di site ini.

Anda juga dapat bergabung kedalam milis C# Indonesia di http://groups.yahoo.com/group/csharp_indo

Komunitas .NET Indonesia (INDC)

Komunitas .NET Indonesia merupakan official dari komunitas-komunitas di Indonesia yang berbasis .NET. Anda bisa bertemu dengan beberapa aktivis komunitas .NET diseluruh wilayah Indonesia.

Anda dapat bergabung kedalam komunitas ini dengan masuk ke url www.netindonesia.net dan dapat juga bergabung ke milis DOTNET