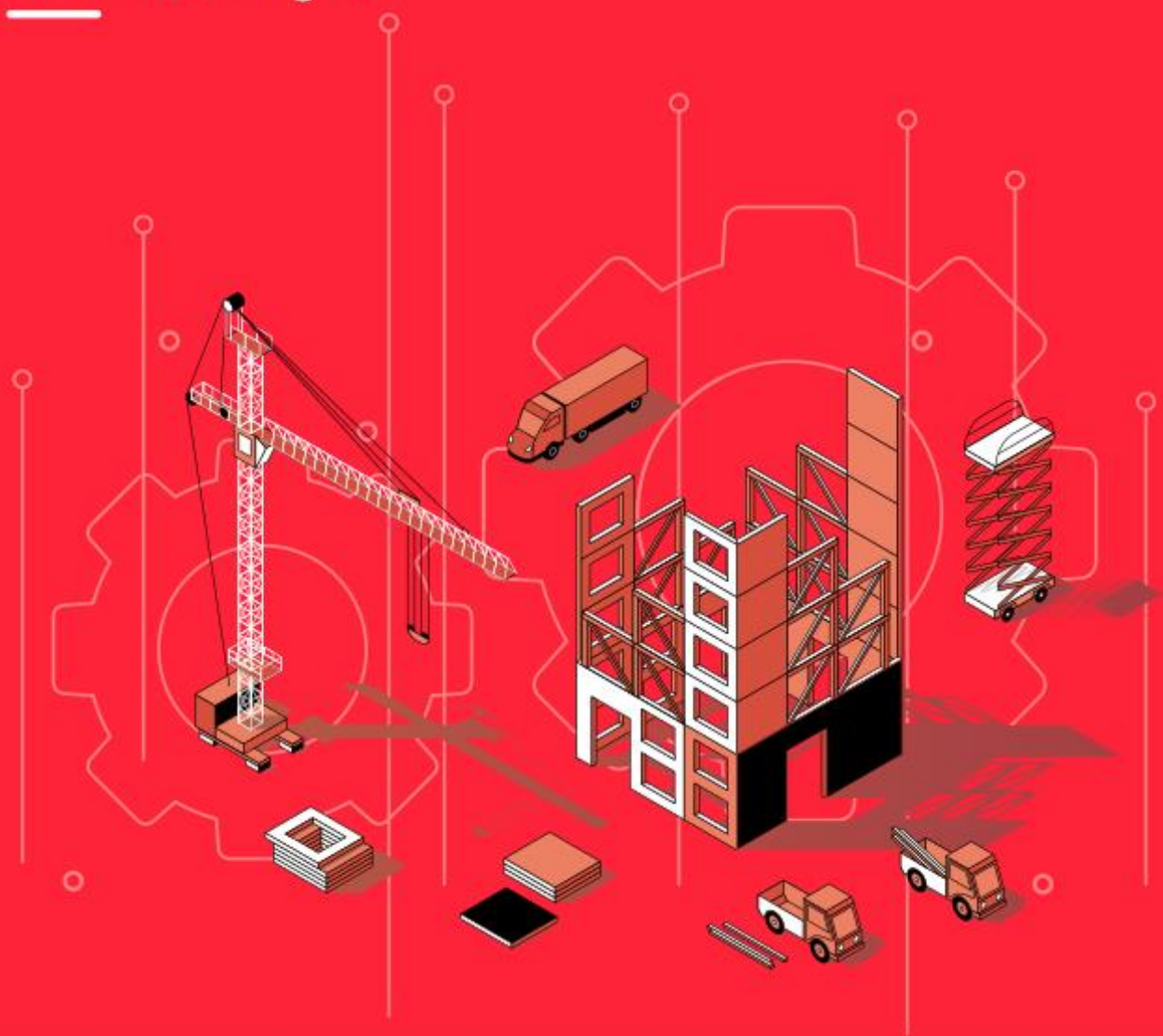


SERI BELAJAR PEMROGRAMAN C#
MahirKoding.ID



MENGUASAI PEMROGRAMAN BERORIENTASI OBJEK DENGAN BAHASA C#

Membangun Aplikasi Software Yang *Maintainable* Dan *Extensible*

DIAN NANDIWARDHANA

Halaman ini sengaja dibiarkan kosong

Menguasai Pemrograman Berorientasi Objek Dengan Bahasa C#

*Membangun Aplikasi Software Yang
Maintainable Dan Extensible*

Dian Nandiwardhana



MahirKoding.ID

Halaman ini sengaja dibiarkan kosong

Daftar Isi

Kata Pengantar.....	i
BAB 1: Paradigma Pemrograman	1
Paradigma pemrograman terstruktur.....	1
Paradigma pemrograman berorientasi objek	4
Alasan menggunakan paradigma berbasis objek.....	6
BAB 2: Empat Pilar Pemrograman Berorientasi objek.....	9
Enkapsulasi (<i>encapsulation</i>).....	9
Abstraksi (<i>abstraction</i>)	10
Pewarisan (<i>inheritance</i>)	11
Polimorfisme (<i>polymorphism</i>)	13
BAB 3: Class	15
Perbedaan antara class dan objek.....	15
Mendeklarasikan sebuah class	16
Class sebagai tipe data.....	17
Membuat objek dari sebuah class (Instansiasi).....	18

BAB 4: Merepresentasikan Karakteristik dan Perilaku Sebuah objek 19

Data Field 19

Kata kunci *const* 20

Kata kunci *readonly* 22

Compile-time constant vs Run-time constant 24

Method 26

Constructor 28

Constructor tanpa parameter (*default constructor*) 29

Constructor berparameter 33

Copy Constructor 36

Kata kunci “*this*” 39

Menggunakan lebih dari satu constructor dalam satu class 45

Constructor overloading 47

Constructor chaining 49

BAB 5: Mengimplementasikan Enkapsulasi Dengan Sebuah Class 59

Mengenal istilah *assembly* di lingkungan .NET 59

Mengakses anggota class	61
public.....	62
private	62
protected	63
internal.....	63
Properti	64
Tipe-tipe properti.....	70
Hindari mendeklarasikan sebuah field sebagai public	71
Auto-implemented property	76
Mengkapsulasi beberapa fungsionalitas dengan utility class.....	77
Ketika objek menjadi ide buruk.....	79
Static class	80
Static members	83
Static constructor	90
Private constructor	95
BAB 6: Mewariskan Karakteristik Dan Perilaku Sebuah Class..	97
Keluarga objek	97
Generalisasi dan spesialisasi	97

Pewarisan dalam pemrograman berorientasi objek	99
Mengimplementasikan konsep pewarisan di C#	102
Pewarisan pada constructor	106
Anggota class dengan akses private tidak bisa diwariskan.....	116
BAB 7: Mengenal Abstract Class Dan Method Overriding.....	122
Abstract class	122
Menyediakan method yang bisa ditimpa (overriding).....	123
Abstract method.....	124
Virtual method	125
Menggantikan method pada base class	126
Method hiding	127
Method overriding	129
BAB 8: Perilaku Polimorfik Dan Ekstensibilitas Software.....	134
Tipe Objek.....	134
Menerapkan polimorfisme	136
Interaksi antar objek.....	141
Antarmuka polimorfik	144
Ekstensibilitas	146

Keterbatasan abstract class sebagai antarmuka polimorfik	150
BAB 9: Interface	155
Interface dan abstraksi	155
Mendefinisikan interface	156
Interface sebagai kontrak	158
Implementasi interface pada sebuah class	159
Antarmuka polimorfik menggunakan interface	161
Mengimplementasikan lebih dari satu interface	164
BAB 10: Studi Kasus: Membuat Simulator Kendaraan.....	167
Mengidentifikasi sistem	167
Mempersiapkan aplikasi di Visual Studio.....	168
Membuat aplikasi konsol.....	168
Membuat class library	172
Menambahkan interface ke dalam class library.....	176
Merubah nama class	178
Menambahkan class ke dalam class library	180
Antarmuka untuk interaksi antar objek.....	182
Mengidentifikasi relasi antar class	184

Pewarisan	184
Asosiasi.....	187
Komposisi.....	190
Mengimplementasikan interface.....	192
Menyembunyikan class dari dunia luar	195
Mengkapsulasi proses instansiasi objek.....	197
Mengonsumsi class library	201
Mengompilasi dan menjalankan program	203
Tantangan	205
Tentang Penulis	I
Daftar Pustaka.....	II

Kata Pengantar

Belajar pemrograman secara mandiri (otodidak) tidak pernah mudah. Terutama, ketika mencari sumber-sumber untuk dipelajari. Saya pun dulu begitu.

Waktu itu, awal tahun 2000-an, saya mengandalkan sumber-sumber yang ada di internet dan buku-buku yang saya beli. Namun, membeli buku berbahasa Indonesia sering kali membuat saya kecewa. Materinya dangkal, tidak sesuai dengan judul bukunya yang luar biasa. Mungkin kebanyakan penulis buku pada waktu itu hanya mengambil materi-materi dari Internet dan diterjemahkan dengan sedikit modifikasi. Entahlah.

Sekarang, meskipun informasi dan materi-materi tentang pemrograman sudah bertebaran bagai jamur di Internet, sayangnya masih didominasi oleh materi-materi berbahasa asing. Materi berbahasa Indonesia, meskipun cukup banyak, masih sedikit yang benar-benar berkualitas. Materi berupa buku pun, tidak banyak berubah.

Beruntung, hampir 10 tahun belakangan ini saya mendapat kemudahan akses untuk mendapatkan buku-buku yang ditulis oleh kalangan profesional di luar negeri. Belum lagi, senior-senior di kantor yang bersedia menjadi mentor membuat proses belajar saya berkembang secara eksponensial.

Akhir tahun 2018 saya berpikir. Apa yang bisa saya lakukan untuk Anda yang ingin belajar pemrograman? Mungkin Anda termasuk yang sulit memahami

materi berbahasa asing. Sementara, materi pemrograman berbahasa Indonesia yang berkualitas masih terbatas. Ketika ada yang berkualitas pun, sudah dipatok dengan harga yang mahal, padahal materinya sangat *basic*.

Akhirnya, saya memutuskan untuk menulis buku elektronik. “Mudah Belajar Pemrograman C#” yang saya rilis awal tahun 2019 adalah buku elektronik pertama saya. Buku kedua ini saya tulis sebagai wujud komitmen saya untuk membantu Anda memulai belajar pemrograman.

Buku ini bukan sekedar referensi sintaks OOP di C#. Sekedar memberikan referensi sering kali berujung kebingungan seperti, mengapa sintaks ini ada sementara kita bisa mendapatkan hasil yang sama ketika memakai sintaks yang lain?

Namun demikian, ini juga bukan buku tentang OODA atau Analisa dan Desain Berorientasi Objek. Hanya saja, saya mencoba memberikan sedikit konsep dan prinsip-prinsip tentang bagaimana mendesain sebuah software yang *maintainable*, *extensible*, dan *flexible*.

Terakhir, saya ingin mengucapkan selamat belajar buat Anda! Semoga dengan mempelajari materi dari buku ini, kedepannya bisa mendatangkan banyak manfaat bagi Anda.

Dian Nandiwardhana

Takarazuka - Agustus 2019

Bab 1

Paradigma Pemrograman

Paradigma pemrograman terstruktur

Dengan paradigma pemrograman terstruktur, sebuah program komputer sebisa mungkin dikembangkan dalam blok-blok dan urutan eksekusi yang logis supaya program tersebut menjadi lebih mudah dipahami dan dipelihara.

Anggaplah Anda biasa melakukan beberapa aktivitas pada saat menjalani rutinitas harian seperti, bangun tidur, mengeluarkan kendaraan dari garasi, mandi, sarapan, memarkir kendaraan, memakai baju, bekerja atau mengikuti pelajaran di sekolah/ kampus, dan lain-lain.

Menggunakan pola pikir terstruktur, Anda harus mampu mengelompokkan aktivitas-aktivitas tersebut berdasarkan aktivitas yang berhubungan dan mengurutkannya sesuai dengan urutan yang logis. Misalnya seperti ini,

Kelompok kegiatan: Persiapan

- 1) Bangun tidur
- 2) Melepas pakaian
- 3) Mandi
- 4) Memakai pakaian
- 5) Sarapan

Kelompok kegiatan: Berangkat ke sekolah/kampus

- 1) Mengeluarkan kendaraan dari garasi
- 2) Mengendarai kendaraan
- 3) Memarkir kendaraan

Kelompok kegiatan: Belajar di sekolah/kampus

- 1) Mengeluarkan catatan dan buku dari dalam tas
- 2) Mencatat materi pelajaran
- 3) Memasukkan catatan dan buku ke dalam tas

Pada susunan di atas, aktivitas-aktivitas yang biasa Anda lakukan dikelompokkan berdasarkan gambaran umum kegiatannya seperti ***Persiapan, Berangkat sekolah/kuliah, dan Belajar di sekolah/kampus.***

Selain itu, aktivitas-aktivitas yang anda lakukan pada setiap kelompok kegiatan di atas, juga diurutkan sesuai dengan urutan yang logis.

Rasa-rasanya tidak wajar kalau Anda mandi terlebih dulu baru melepas pakaian. Anda juga tidak mungkin mengendarai kendaraan terlebih dulu baru kemudian mengeluarkannya dari garasi. Atau, Mencatat materi pelajaran tanpa sebelumnya mengeluarkan buku catatan dari dalam tas Anda.

Dengan menyusun daftar aktivitas dan mengelompokkannya berdasarkan aktivitas yang berhubungan, Anda dapat mengubah aktivitas pada satu

kelompok kegiatan tanpa mengubah urutan kegiatan harian Anda secara garis besar.

Misalnya, Anda ingin mengubah daftar aktivitas yang Anda lakukan pada saat berangkat ke sekolah atau ke kampus menjadi seperti berikut,

- 1) Mengecek informasi lalu lintas dan cuaca
- 2) Memutuskan untuk mengendarai kendaraan pribadi atau transportasi umum
- 3) Jika Anda memutuskan untuk mengendarai motor, keluarkan kendaraan Anda dari garasi dan kendarai kendaraan Anda ke sekolah atau kampus.
- 4) Jika tidak, jalan ke halte bus atau stasiun kereta, lalu naik bus atau kereta untuk berangkat ke sekolah atau kampus.

Secara umum, urutan kegiatan harian Anda akan tetap sama meskipun Anda telah mengubah jenis aktivitas yang Anda lakukan pada saat berangkat sekolah / kuliah. Yaitu, secara umum, Anda melakukan **Persiapan**, lalu **Berangkat sekolah / kuliah**, dan sesampainya di sekolah atau kampus, Anda **Belajar di sekolah / kampus**.

Dalam pemrograman terstruktur, Anda mengelompokkan setiap baris instruksi (aktivitas) yang berhubungan ke dalam sebuah *method* atau *function* (kelompok kegiatan).

Paradigma pemrograman berorientasi objek

Berbeda dengan paradigma pemrograman terstruktur yang menitikberatkan pada sebuah **mekanisme atau aktivitas** untuk membuat sebuah instruksi, paradigma pemrograman berorientasi objek menganjurkan Anda untuk memodelkan instruksi-instruksi pada program komputer dengan mengacu pada **data** yang akan dimanipulasi oleh instruksi-instruksi tersebut dan sekaligus menyimpannya bersama-sama sebagai sebuah komponen.

Untuk lebih memahami konsep tersebut, coba bayangkan Anda memiliki sebuah buku alamat yang menyimpan beberapa informasi atau data mengenai teman Anda seperti,

- Nama
- Alamat
- Nomor HP

Apa yang kira-kira bisa Anda lakukan terhadap buku alamat ini?

Tentunya Anda bisa **mencari** detail informasi mengenai teman Anda seperti nomor HP-nya. Lalu, Anda juga bisa **menambahkan** informasi ke dalam buku alamat tersebut ketika Anda bertemu dengan teman baru. Selain itu, Anda juga bisa **menghapus** sebuah informasi dari dalam buku alamat tersebut ketika Anda perlu *move-on* setelah putus dari kekasih Anda.

Dengan paradigma pemrograman berorientasi objek, Anda bisa membuat sebuah komponen software yang dapat menyimpan dan mengelola

informasi/data mirip dengan buku alamat tadi.

Dengan membuat buku alamat tadi sebagai sebuah komponen software, Anda bisa menggunakan kembali komponen buku alamat yang telah Anda buat tersebut. Jadi, ketika Anda ingin membuat sistem atau aplikasi lain yang juga membutuhkan fitur serupa, Anda tinggal menggunakannya lagi tanpa harus menulis dan mengkompilasi ulang kode program yang sama.

Misalnya, Anda diminta untuk membuat sebuah aplikasi untuk sistem pengelolaan perpustakaan. Salah satu fitur dari aplikasi tersebut adalah bisa menyimpan dan mengelola informasi dari pengguna perpustakaan. Dengan kebutuhan seperti itu, Anda kemudian membuat sebuah komponen yang mirip dengan sebuah buku alamat.

Di kesempatan yang lain, Anda diminta untuk membuat aplikasi untuk sistem pengelolaan pengadaan oleh sebuah perusahaan. Salah satu fitur dari aplikasi tersebut adalah bisa menyimpan dan mengelola informasi dari supplier-supplier perusahaan tersebut.

Karena sebelumnya Anda pernah membuat sebuah komponen software yang bisa menyimpan dan mengolah data seperti sebuah buku alamat, pada saat menulis kode program untuk aplikasi sistem pengelolaan pengadaan, Anda tidak perlu lagi menulis dan mengkompilasi ulang kode program yang sama untuk sebuah fitur yang sama. Anda tinggal menggunakan komponen software buku alamat tadi untuk aplikasi tersebut.

Anda akan semakin memahami konsep ini setelah Anda mempelajari apa itu *Dynamic Link Library* atau DLL yang akan dibahas pada Bab 5.

Alasan menggunakan paradigma berbasis objek

Dari pengalaman saya sebagai *software developer* yang bekerja dalam sebuah tim dan juga pengalaman melakukan *review* kode program yang dikerjakan oleh pihak ketiga, apapun metodologi yang dipakai untuk mengembangkan software, permasalahannya seringkali bukan terletak pada penulisan kode program di awal pengembangan aplikasi. Namun, permasalahan sering kali timbul dari:

- 1) *Maintainability* yang buruk. *Maintainability* didefinisikan sebagai kemudahan suatu software untuk dimengerti, diperbaiki, diadaptasi dan/atau dikembangkan. Dengan *maintainability* yang buruk, sebuah sistem akan susah dimengerti dan direvisi ketika ada permintaan untuk mengganti atau menambah sebuah fungsi atau fitur.
- 2) Dalam artikelnya di jurnal *IEEE Software*, Robert L. Glass, menyatakan bahwa biaya *maintenance* biasanya menghabiskan 40% sampai 80% (rata-rata 60%) biaya pengembangan software. Oleh karena itu fase *maintenance* merupakan fase yang perlu mendapat perhatian khusus. Biaya tersebut dikeluarkan justru setelah sebuah software dirilis. Setelah rilis, tinggi kemungkinan fungsi dan fitur sebuah software perlu

diadaptasi untuk memenuhi perubahan kebutuhan sebuah organisasi yang menggunakan aplikasi tersebut. Oleh karena itu, untuk meminimalisir biaya *maintenance*, seorang software engineer harus benar-benar memastikan bahwa aplikasi software yang dikembangkannya mudah untuk di-*maintain* setelah dirilis.

Permasalahan-permasalahan ini sama halnya ketika Anda ingin membangun sebuah rumah. Anda bisa saja memanggil seorang tukang bangunan dan menyampaikan keinginan-keinginan Anda. Seorang tukang bangunan pastinya tahu betul bagaimana mengaduk semen, memasang batu bata, dan lain-lain.

Rumah yang dibangun mungkin nantinya bisa berfungsi sebagai mestinya. Namun, tanpa perencanaan yang matang pada saat fase awal pembangunan rumah Anda, ketika kedepannya Anda ingin mengalih fungsikan dan/atau menambahkan sebuah ruangan, tentunya pekerjaan ini akan menjadi lebih sulit dan pastinya memakan biaya yang lebih banyak.

Berbeda ketika Anda menyampaikan keinginan-keinginan Anda pada seorang arsitek yang tahu betul bagaimana mendesain sebuah rumah yang sesuai dengan kebutuhan Anda.

Seorang arsitek akan menganalisa apa saja kebutuhan-kebutuhan Anda saat ini dan kemungkinan perubahan kebutuhan di masa yang akan datang. Dari hasil analisa ini, dia akan memulai membuat perencanaan desain yang

matang agar rumah Anda nantinya memiliki *maintainability* yang tinggi.

Dalam dunia pengembangan software, paradigma pemrograman berbasis objek ini bertujuan untuk membantu mengatasi permasalahan-permasalahan tersebut dengan menitikberatkan pada **tugas-tugas analisa dan desain** di fase awal pengembangan aplikasi software, agar Anda bisa memastikan bahwa software yang dikembangkan mempunyai *maintainability* yang tinggi (*robust* dan *maintainable*).

Bab 2

Empat Pilar Pemrograman Berorientasi objek

Enkapsulasi (*encapsulation*)

objek adalah sesuatu yang memiliki sebuah karakteristik seperti “tinggi”, “berat”, dan “lebar” atau kondisi (*state*) seperti “senang”, “sedih”, “panas”, “dingin”, dan lain-lain. Selain itu, sebuah objek juga memiliki mekanisme atau perilaku seperti “berlari”, “berjalan”, dan lain-lain.

Di dunia nyata, interaksi antar objek pada umumnya tidak bisa dihindari. Namun yang jelas, pada saat sebuah objek berinteraksi dengan objek yang lain, karakteristik atau kondisi dari sebuah objek tidak bisa semena-mena diubah secara langsung oleh objek yang lain, kecuali dilakukan melalui sebuah interaksi yang kemudian mengaktifkan mekanisme internal dari objek tersebut.

Misalnya, Anda sedang jatuh cinta dengan seseorang. Anda tentu saja tidak bisa dengan semena-mena mengganti perasaan (kondisi) gebetan dari yang semula “biasa” saja menjadi “cinta”. Anda mungkin terlebih dahulu perlu berinteraksi, misalnya berkenalan dan mengobrol, untuk membuat dia jatuh cinta juga kepada Anda (mengaktifkan mekanisme internal). Begitu, bukan?

Mekanisme internal yang membuat gebetan Anda bisa jatuh cinta kepada Anda ini sangat kompleks dan tersembunyi dari jangkauan Anda. Inilah pentingnya enkapsulasi. Bayangkan saja apa jadinya interaksi antar objek di dunia nyata tanpa adanya enkapsulasi.

Dalam pemrograman berorientasi objek, enkapsulasi adalah sebuah cara untuk mengumpulkan data-data dan beberapa fungsionalitas dalam sebuah unit tunggal dan sekaligus membuat mereka aman terhadap interaksi langsung dari luar.

Abstraksi (*abstraction*)

Pada saat menggunakan perangkat *smartphone*, apakah Anda perlu mengetahui bagaimana masing-masing komponen di dalam perangkat *smartphone* Anda bekerja?

Jika iya, Anda pasti akan dibuat pusing setiap kali sebuah *vendor smartphone* mengeluarkan seri *smartphone* terbarunya dengan komponen-komponen dan mekanisme internal yang berbeda.

Untungnya, hal ini tidak pernah terjadi.

Selama Anda mengerti bagaimana layar monitor, tombol *power*, dan tombol *volume* pada sebuah perangkat *smartphone* bekerja, Anda akan tetap bisa menggunakan perangkat Anda dengan mudah, meskipun komponen-komponen dan mekanisme internalnya sudah jauh berubah. Hal ini

dikarenakan perangkat *smartphone* Anda merupakan abstraksi pada tingkatan yang lebih tinggi (*high level abstraction*) dari sebuah sistem yang sangat kompleks.

Pada saat mendesain komponen *software* yang ingin Anda buat, mengidentifikasi elemen-elemen tingkat tinggi apa saja yang diperlukan untuk berinteraksi dengan komponen tersebut menjadi hal yang sangat penting untuk mendapatkan abstraksi yang baik. Sebuah perangkat *smartphone*, contohnya, memerlukan elemen-elemen tingkat tinggi seperti layar monitor, tombol *power*, tombol *volume*, dan sistem operasi agar pengguna perangkat tersebut bisa mengoperasikannya dengan mudah.

Dengan menentukan abstraksi untuk komponen *software* Anda, apabila nantinya Anda melakukan baik perubahan kecil maupun besar pada mekanisme internal komponen *software* tersebut, pengguna komponen Anda tetap bisa menggunakannya seperti biasa.

Pewarisan (*inheritance*)

objek yang berada dalam satu kelompok kategori biasanya memiliki karakteristik dan perilaku yang sama satu sama lainnya.

Misalnya, sebuah mobil dan sebuah motor merupakan dua objek yang sangat berbeda. Meskipun demikian, mobil dan motor sama-sama memiliki roda, menggunakan bahan bakar untuk mesinnya, perlu menghidupkan mesinnya terlebih dulu sebelum bisa dikendarai, dan kemudian mematikan mesinnya

ketika tidak digunakan.

Beberapa kesamaan karakteristik dan perilaku dari mobil dan motor itu sejatinya karena **“mobil”** dan **“motor”** sama-sama merupakan sebuah **“kendaraan bermotor”**.

Oleh karena itu, semua objek yang termasuk dalam kelompok kategori kendaraan bermotor, **“mewarisi”** karakteristik dan perilaku yang terdapat pada kendaraan bermotor secara umum.

Dalam dunia pemrograman berbasis objek, dengan menggunakan konsep pewarisan/*inheritance*, Anda dapat membuat sebuah *class* bernama “kendaraan bermotor” yang memiliki karakteristik dan perilaku yang umum atau biasa terdapat pada semua jenis kendaraan yang termasuk dalam kelompok kategori “kendaraan bermotor”.

Lalu, ketika Anda membuat *class* seperti “mobil” dan “motor”, Anda tinggal membuat *class* tersebut **“mewarisi”** karakteristik dan perilaku dari *class* “kendaraan bermotor” tersebut.

Dengan demikian, *class* “mobil” dan “motor” akan memiliki karakteristik dan perilaku yang sama persis dengan yang dimiliki oleh *class* “kendaraan bermotor”. Selanjutnya, Anda tinggal melengkapi karakteristik dan perilaku spesifik yang biasanya hanya terdapat pada sebuah mobil untuk *class* “mobil”, dan juga melengkapi karakteristik dan perilaku spesifik yang biasanya hanya terdapat pada sebuah motor untuk *class* “motor”.

Polimorfisme (*polymorphism*)

Ketika sebuah objek dapat berganti peranan sesuai dengan kondisi yang diberikan, artinya objek tersebut memiliki sifat polimorfisme. Polimorfisme sendiri adalah kemampuan sebuah objek untuk berperan dalam beberapa bentuk.

Anda sendiri sebenarnya memiliki sifat polimorfisme. Bukan berarti Anda bisa berubah menjadi seekor kodok dan kemudian berubah lagi menjadi seekor burung merpati. Namun, Anda mempunyai sifat polimorfisme dalam konteks peran yang Anda lakukan sesuai kondisi yang sedang Anda jalankan.

Misalnya, ketika Anda sedang berada di kampus, Anda sedang berperan sebagai seorang mahasiswa. Ketika sedang berlalu lintas, Anda sedang berperan sebagai pengguna jalan. Ketika Anda sedang bekerja paruh waktu di sebuah restoran, Anda berperan sebagai pekerja restoran.

Perilaku Anda ketika berperan sebagai mahasiswa tentunya berbeda dengan ketika Anda berperan sebagai pengguna jalan atau pekerja restoran.

Konsep polimorfisme ini di dalam dunia pemrograman, contohnya, bisa Anda gunakan ketika Anda membuat sebuah *class* untuk melakukan pengukuran terhadap beberapa bidang geometri, seperti membuat satu *class* yang bisa mengukur baik luasan sebuah segitiga, persegi panjang, lingkaran, dan lain-lain.

Apabila Anda ingin mengukur luasan segitiga, *class* tersebut akan berperan

mengukur luasan segitiga. Begitu pula apabila Anda ingin mengukur luasan lingkaran, *class* tersebut akan berperan mengukur luasan lingkaran.

Jadi, meskipun rumus untuk menghitung luasan segitiga, persegi panjang, dan lingkaran itu berbeda-beda. Dengan membuat sebuah *class* yang mengadopsi konsep polimorfisme, Anda dapat menggunakan *class* tersebut untuk mengukur berbagai jenis bidang geometri.

Bab 3

Class

Perbedaan antara class dan objek

Seperti yang sudah saya jelaskan di buku “Mudah Belajar Pemrograman C#”, sebuah *class* adalah sebuah cetak biru (*blueprint*) yang menjadi acuan untuk membuat sebuah objek.

Namun demikian, masih ada beberapa orang yang menganggap bahwa sebuah *class* dan objek adalah satu istilah yang bisa saling menggantikan. Anggapan ini tentunya kurang tepat, karena istilah *class* dan objek adalah dua istilah yang berbeda walaupun sangat erat kaitannya satu sama lainnya.

Untuk lebih memahami apa bedanya *class* dan objek, coba Anda bayangkan sebuah sistem pengelolaan rekening di sebuah Bank.

Apa saja informasi yang perlu disimpan dan bagaimana saldo pada sebuah rekening dikelola, harus **dimodelkan** terlebih dahulu agar nantinya satu rekening dengan rekening yang lain memiliki struktur informasi dan cara pengelolaan yang sama. Model (cetak biru) dari sebuah rekening ini dalam pemrograman disebut dengan *class*.

Sementara, setiap seseorang membuka rekening baru di bank tersebut,

sebuah objek rekening dari *class* rekening tersebut akan dibuat. Setiap objek nantinya akan memiliki informasi yang unik.

Maksudnya, jika Anda yang membuka rekening baru di bank tersebut, objek tersebut hanya akan memuat informasi yang secara khusus diberikan kepada Anda. Objek rekening yang dibuat untuk rekening Anda ini akan berisi informasi seperti nama, alamat, dan nomor NPWP Anda serta sebuah nomor rekening yang unik.

Ketika ada orang lain yang membuka rekening di bank tersebut, sebuah objek rekening yang berbeda dari *class* rekening yang sama juga akan dibuat. Kali ini, meskipun memiliki struktur informasi yang sama, objek tersebut memuat informasi yang berbeda dari objek rekening yang dibuat untuk Anda.

Dalam pemrograman Bahasa C#, objek disebut ***instance*** dari sebuah *class*. Sedangkan proses pembuatan sebuah objek dari sebuah *class* disebut dengan ***instantiation*** (instansiasi).

Mendeklarasikan sebuah class

Untuk mendeklarasikan sebuah *class*, gunakan kata kunci **class** seperti pada contoh berikut ini.

```
class NamaClass
{
    // Anggota penyusun class (class members) untuk menentukan
    // karakteristik dan perilaku class yang dideklarasikan.
}
```

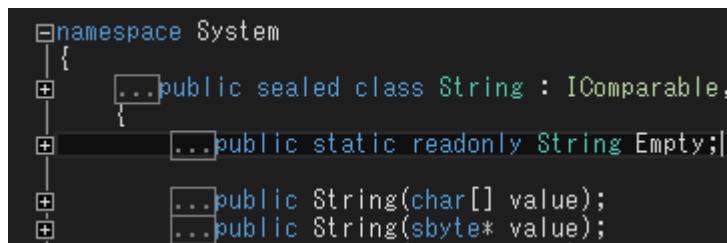
Class sebagai tipe data

Sampai di sini, Anda pastinya sudah familiar dengan tipe data. C# sendiri menyediakan beberapa tipe data yang bisa langsung Anda gunakan, seperti `int`, `double`, `bool`, `string`, dan lain-lain. Tipe data ini disebut dengan *pre-defined* atau *built-in data type*.

Ketika Anda mendeklarasikan sebuah *class*, maka sebenarnya Anda sedang membuat sebuah tipe data baru yang nantinya bisa digunakan untuk mendeklarasikan sebuah variabel seperti potongan kode program berikut ini.

```
NamaClass namaVar; // Mendeklarasikan sebuah variabel
```

Tipe data `string` sendiri sebenarnya adalah sebuah *class* di dalam *namespace system* milik .NET seperti yang bisa Anda lihat pada gambar 3.1.



```
namespace System
{
    ...public sealed class String : IComparable,
    {
        ...public static readonly String Empty;
    }
    ...public String(char[] value);
    ...public String(sbyte* value);
}
```

Gambar 3.1

Karena Anda bisa membuat tipe data Anda sendiri dengan sebuah *class*, maka dalam pemrograman berorientasi objek, *class* bisa disebut dengan *user-defined data type*.

Membuat objek dari sebuah class (Instansiasi)

Agar Anda bisa menggunakan karakteristik dan perilaku yang Anda definisikan dalam sebuah *class*, Anda perlu membuat sebuah *instance* dari *class* tersebut. Sebuah *instance* dari sebuah *class* disebut dengan objek.

Untuk membuat sebuah *instance* dari sebuah *class*, gunakan kata kunci `new`, seperti pada potongan kode program berikut ini.

```
// Instansiasi sebuah class
// membuat sebuah objek bertipe Mobil dari tipe Mobil
Mobil mobil = new Mobil();
```

Ketika Anda menginstansiasi sebuah *class* dengan cara seperti di atas, sebenarnya Anda melakukan dua hal, yaitu:

- 1) Anda membuat sebuah objek baru dari *class* `Mobil` dan menyimpannya di memori *heap*.

```
new Mobil()
```

- 2) Anda membuat sebuah referensi objek di dalam memori *stack* yang bertipe data `Mobil` dengan nama `mobil` yang menunjuk alamat memori *heap* di mana objek baru dari *class* `Mobil` ini disimpan.

```
Mobil mobil
```

Setelah Anda menginstansiasi sebuah objek, Anda dapat menggunakan karakteristik dan perilaku yang didefinisikan pada sebuah *class*.

Bab 4

Merepresentasikan Karakteristik dan Perilaku Sebuah objek

Di bab 3 telah dijelaskan bahwa sebuah *class* nantinya akan menentukan karakteristik dan perilaku setiap objek dari *class* tersebut.

Karakteristik dan perilaku tersebut dapat direpresentasikan dengan mendefinisikan beberapa anggota penyusun *class* (*class members*) seperti *field*, *method*, *property*, dan *event* di dalam sebuah *class*.

Pada bab ini, Anda akan mempelajari apa fungsi dari setiap anggota penyusun *class* dan bagaimana menambahkannya ke dalam sebuah *class*.

Data Field

Sebuah *data field*, atau biasa disebut *field*, sejatinya adalah sebuah konstanta atau variabel dengan tipe data apapun yang dideklarasikan langsung pada sebuah *class*. *Field* digunakan untuk merepresentasikan karakteristik atau kondisi sebuah objek yang dibuat dari *class* tersebut.

Field biasanya menyimpan data yang harus bisa diakses oleh **lebih dari satu** *method* yang didefinisikan di dalam sebuah *class*. Apabila Anda menginginkan sebuah variabel yang hanya digunakan oleh satu buah *method*

saja, lebih baik Anda mendeklarasikan variabel tersebut sebagai variabel lokal di dalam *method* yang menggunakannya saja.

Untuk mendeklarasikan sebuah variabel sebagai *field* di dalam sebuah *class*, Anda harus mendeklarasikan variabel tersebut di dalam blok *class* seperti pada potongan kode program berikut ini.

```
class Mobil
{
    // Mendeklarasikan variabel sebagai field dari class Mobil
    public string pabrikan;
    public string tipe;
    public int thnPembuatan;
}
```

Jika Anda perhatikan, cara mendeklarasikan sebuah variabel sebagai *field* tidak berbeda dengan mendeklarasikan sebuah variabel pada umumnya. Kata kunci **public** adalah *access modifier* yang akan Anda pelajari di bab 5.

Kata kunci const

Dalam pemrograman C#, selain mendeklarasikan variabel, Anda juga bisa mendeklarasikan konstanta sebagai *field*. Konstanta merupakan suatu nilai tetap.

Untuk mendeklarasikan sebuah konstanta, Anda bisa menggunakan kata kunci **const** di depan tipe data dan nama konstanta yang dideklarasikan, seperti contoh berikut ini.

```
public const double Pi = 3.14159;
```


Anda bisa menggunakan kata kunci **const** dengan variabel yang bernilai sebuah angka, **boolean**, **string**, atau dengan referensi null.

Ketika Anda mendeklarasikan sebuah *field* dengan kata kunci **const**, Anda juga diharuskan untuk menetapkan nilai awal atau menginisialisasi *field* tersebut. Apabila Anda mendeklarasikan konstanta Pi menjadi seperti kode program di bawah ini, *compiler* akan memberi Anda pesan error: *‘A const field requires a value to be provided’*.

```
public const double Pi; // error
```

Sekali Anda mendeklarasikan sebuah konstanta, maka Anda tidak bisa mengganti nilai dari konstanta tersebut.

```
class Lingkaran
{
    public const double Pi = 3.14159;

    public void GantiPi()
    {
        Pi = 3.14; // Menghasilkan compiler error
    }
}
```

Apabila Anda mencoba mengkompilasi kode program di atas, *compiler* akan memberi Anda pesan error: *‘The left-hand side of an assignment must be a variable, property or indexer’*.

Sebuah *field* yang dideklarasikan dengan kata kunci **const** merupakan **static field** secara default. Oleh karena itu, Anda bisa mengakses *field* Pi

tanpa harus terlebih dahulu membuat objek dari *class* `Lingkaran`. Cukup dengan nama *class*-nya saja, seperti pada kode program di bawah ini.

```
class Lingkaran
{
    public const double Pi = 3.14159;
}

class Program
{
    static void Main(string[] args)
    {
        int r = 0;
        double luasLingkaran = 0;
        // Mengakses konstanta Pi dari class Program
        luasLingkaran = Lingkaran.Pi * r * r;
    }
}
```

Di bab 5, Anda akan mempelajari apa itu ***static***, bagaimana menerapkannya pada sebuah *class* dan anggotanya, serta pada skenario seperti apa Anda perlu menggunakan kata kunci ***static***.

Kata kunci ***readonly***

Sebuah *field* yang dideklarasikan dengan kata kunci ***readonly*** juga akan menjadi sebuah konstanta, sama seperti ketika dideklarasikan dengan kata kunci ***const***. Sekali nilainya sudah ditetapkan, maka nilainya tidak lagi bisa diubah.

Kode program berikut ini mendemonstrasikan cara untuk mendeklarasikan sebuah konstanta dengan kata kunci ***readonly***.

```
public readonly int _ID = 2014007350;
```

Berbeda dengan menggunakan kata kunci **const**, pada saat Anda menggunakan kata kunci **readonly**, Anda tidak diharuskan untuk menginisialisasi *field* tersebut. Jadi, kode program di bawah ini tidak akan menghasilkan error.

```
public readonly int _ID;
```

Namun, ketika Anda mendeklarasikan *read-only field* seperti kode program di atas, maka Anda harus menginisialisasi *field* tersebut **di dan hanya di** dalam *constructor* seperti pada kode program di bawah ini.

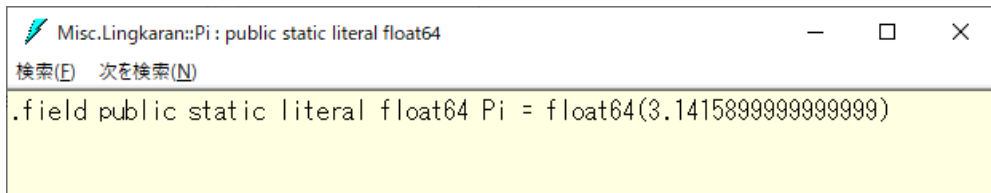
```
class ReadOnlyField
{
    public readonly int _ID;
    public ReadOnlyField(int id)
    {
        _ID = id;
    }
}
```

Constructor sendiri adalah sebuah *method* di dalam sebuah *class* yang akan dieksekusi pada saat sebuah *instance* dibuat dari *class* tersebut. `ReadOnlyField()` adalah *constructor* dari *class* `ReadOnlyField`. (*Constructor* akan Anda pelajari selanjutnya di bab ini)

Compile-time constant vs Run-time constant

Menggunakan kata kunci ***const*** dan ***readonly*** akan sama-sama menghasilkan sebuah konstanta. Bedanya, ***const*** adalah *compile-time constant*. Sedangkan ***readonly*** adalah *run-time constant*.

Yang dimaksud dengan *compile-time constant* adalah, ketika program Anda dikompilasi, *compiler* akan mengevaluasi nilai yang diberikan pada sebuah konstanta. Kemudian menetapkan nilai tadi sebagai nilai dari konstanta tersebut.



Gambar 4.1

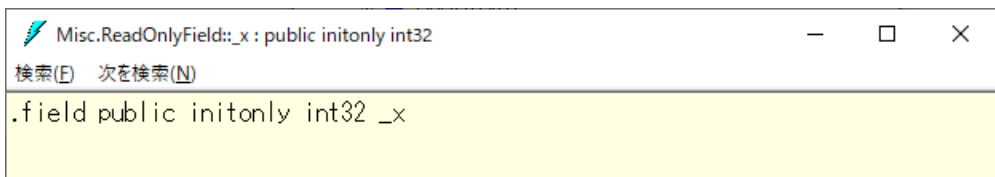
Gambar 4.1 di atas adalah kode CIL (*Common Intermediate Language*) yang dihasilkan oleh *compiler* ketika mengkompilasi konstanta Pi pada kode program sebelumnya. Di situ Anda bisa melihat nilai 3.14158999 ditetapkan sebagai nilai dari *field* Pi. Selain itu, Anda juga bisa melihat bahwa konstanta Pi adalah sebuah *static field*.

Sedangkan yang dimaksud dengan *run-time constant* adalah, *compiler* baru akan mengevaluasi nilai konstantanya pada saat *run-time*, yaitu ketika sebuah program sedang dijalankan. Untuk lebih memahaminya, coba perhatikan kode program di bawah ini. Lalu perhatikan juga kode CIL yang

dihasilkan oleh *compiler* setelah mengkompilasi kode program tersebut.

```
class ReadOnlyField
{
    public readonly int _x = 10;
}
```

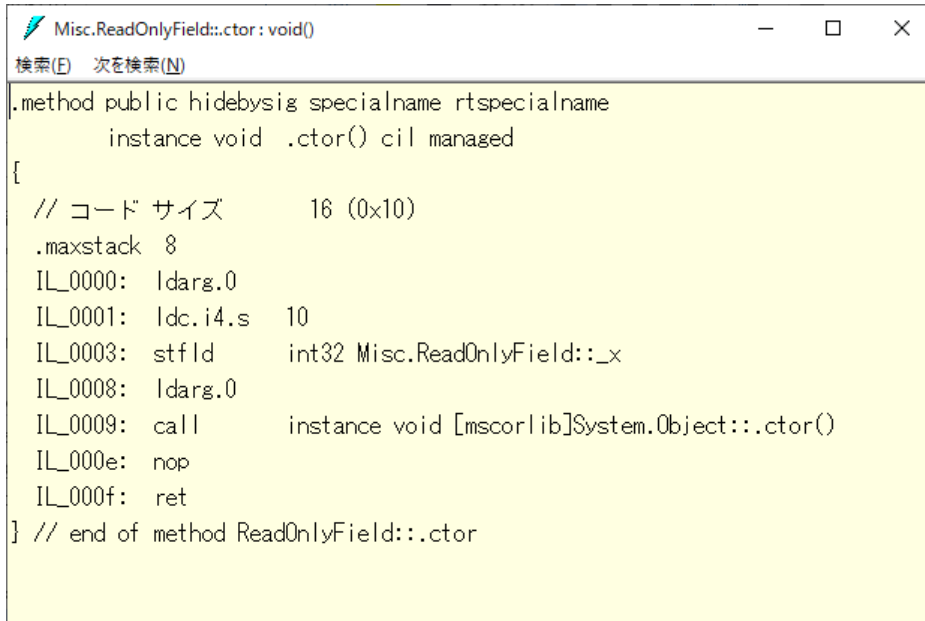
Jika Anda cukup teliti, Ada yang cukup menarik pada kode CIL yang dihasilkan oleh *compiler* pada Gambar 4.2. Pada kode program di atas, bukankah nilai *field* `_x` sudah diinisialisasi dengan angka 10? Tapi mengapa ketika melihat kode CIL, sepertinya *compiler* tidak menetapkan angka 10 ke `_x`?



Gambar 4.2

Jawaban sederhananya, karena `_x` merupakan *run-time constant*, maka nilainya baru akan dievaluasi dan ditetapkan pada saat *run-time*. Bukan pada saat kompilasi.

Jawaban ini akan menjadi lebih jelas apabila Anda memperhatikan Gambar 4.3. yang menampilkan kode CIL hasil dari mengkompilasi *constructor* pada *class* `ReadOnlyField`.



Gambar 4.3

Sekarang, Anda bisa melihat *field* `_x` dan juga angka 10 (`IL_0001: ldc.i4.s 10`). Untuk sementara Anda tidak perlu risau dengan instruksi-instruksi dari kode CIL di atas. Intinya, ketika *constructor* pada *class* `ReadOnlyField` di atas dieksekusi saat *run-time*, angka 10 kemudian ditetapkan ke *field* `_x`.

Sekalinya *field* `_x` sudah ditetapkan nilainya, maka Anda tidak bisa lagi merubah nilainya.

Method

Untuk merepresentasikan perilaku sebuah objek yang dibuat dari sebuah *class*, Anda dapat mendefinisikan beberapa *method* di dalam *class* tersebut.

Seperti yang sudah saya jelaskan di buku “Mudah Belajar Pemrograman C#”, sebuah *method* berisi instruksi-instruksi yang membentuk sebuah mekanisme untuk menjalankan operasi tertentu.

Anda dapat menambahkan beberapa *method* ke dalam sebuah *class* seperti pada contoh kode program berikut ini.

```
class Mobil
{
    // field
    public bool statusMesin;

    // method
    public bool MenghidupkanMesin()
    {
        // instruksi untuk proses menghidupkan mesin mobil
        Console.WriteLine("Menghidupkan mesin mobil");
        // true = ON
        statusMesin = true;
        return statusMesin;
    }

    // method
    public bool MematikanMesin()
    {
        // instruksi untuk proses menghidupkan mesin mobil
        Console.WriteLine("Mematikan mesin mobil");

        // false = OFF
        statusMesin = false;
        return statusMesin;
    }
}
```

Constructor

Dalam pemrograman C#, *constructor* merupakan sebuah *method* khusus yang akan dipanggil pada saat sebuah objek (*instance*) dari sebuah *class* dibuat. *Constructor* mempunyai nama atau *identifier* yang sama persis dengan *identifier class* -nya.

Anda bisa memanfaatkan *constructor* untuk menginisialisasi beberapa *field* yang membutuhkan nilai *default*. Dengan kata lain, *constructor* diperlukan untuk “membangun” (*construct*) objek dari sebuah *class*.

Pada dasarnya sebuah *class* akan memiliki minimal sebuah *constructor*. Jadi, meskipun Anda tidak mendefinisikan sebuah *constructor* secara eksplisit, secara otomatis *compiler* akan membuat *default constructor* dari *class* yang Anda deklarasikan.

Meskipun *constructor* merupakan sebuah *method*, sebuah *constructor* tidak bisa memiliki tipe nilai balik (*return type*) apapun, bahkan **void** sekalipun.

Di bawah ini adalah potongan kode program untuk mendefinisikan sebuah *constructor* secara eksplisit di dalam sebuah *class*.

```
public class Mobil
{
    // Constructor memiliki nama/identifier yang
    // sama dengan class-nya yaitu “Mobil”
    public Mobil()
    {
        // Kode program
    }
}
```



```
}

```

Pada kode program di atas, kita mendeklarasikan sebuah *class* yang diberi nama `Mobil`. Di dalamnya, kita juga mendefinisikan sebuah *method* dengan nama yang sama persis dengan nama *class*-nya, yaitu `Mobil()`. *Method* `Mobil()` inilah yang merupakan *constructor* dari *class* `Mobil`.

Dalam pemrograman C#, ada beberapa tipe *constructor* yang bisa Anda gunakan, yaitu:

- 1) *Default constructor*
- 2) *Parameterized constructor*
- 3) *Copy constructor*
- 4) *Static constructor*
- 5) *Private constructor*

Pada bab ini, saya hanya akan membahas tiga jenis *constructor* saja, yaitu *default constructor*, *parameterized constructor*, dan *copy constructor*. Selanjutnya, *static constructor* dan *private constructor* akan saya bahas di Bab 5.

Constructor tanpa parameter (*default constructor*)

Apabila Anda mendefinisikan sebuah *constructor* tanpa menggunakan parameter sama sekali, maka *constructor* tersebut disebut dengan *default constructor*.

Coba perhatikan kode program berikut ini.

```
using System;

namespace KodeProgram_4_1
{
    class Mobil
    {
        public string pabrik;
        public string model;

        // Default Constructor
        public Mobil()
        {
            pabrik = "Toyota";
            model = "Hatchback";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Membuat objek "mobil" dari class "Mobil" tanpa parameter
            // Constructor "Mobil()" akan dipanggil secara otomatis
            Mobil mobil = new Mobil();
            Console.WriteLine(mobil.pabrik);
            Console.WriteLine(mobil.model);
            Console.WriteLine("\nTekan Enter untuk keluar..");
            Console.ReadLine();
        }
    }
}
```

Pada kode program 4-1 di atas, kita mendeklarasikan dua buah variabel sebagai *field* dari class `Mobil`, yaitu `pabrik` dan `model` yang masing-masing bertipe data `string`.

Di dalam *constructor* `Mobil()`, kita memberi instruksi agar *field* pabrikan diberi nilai “Toyota” dan *field* model diberi nilai “Hatchback”.

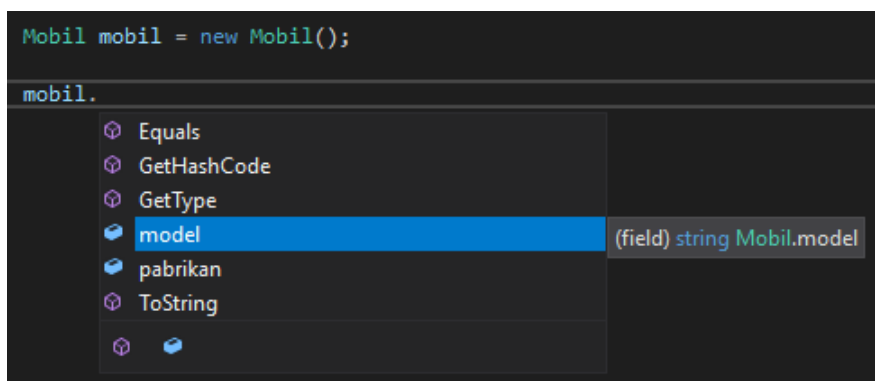
Pada saat kita membuat objek mobil dari *class* `Mobil`,

```
Mobil mobil = new Mobil();
```

kode `new Mobil()` akan memanggil *constructor* `Mobil()` sehingga nilai dari *field* pabrikan dan model akan ditetapkan.

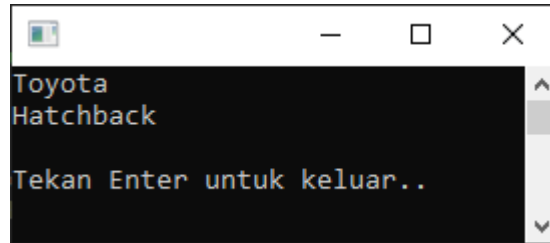
Untuk mengakses nilai sebuah *field* dari sebuah objek, Anda dapat menuliskan nama objek tersebut diikuti dengan tanda titik lalu diikuti dengan nama *field* dari objek tersebut.

Apabila Anda menggunakan Visual Studio, pada saat Anda menambahkan tanda titik setelah nama sebuah objek, fitur *intellisense* pada Visual Studio akan menampilkan semua *field* (dan juga *method*) yang bisa diakses oleh objek tersebut.



Gambar 4.4

Apabila kode program 4-1 Anda jalankan, Anda akan mendapatkan tampilan seperti pada gambar 4.5.



Gambar 4.5

Menggunakan *default constructor* artinya anda menetapkan nilai pasti (*fixed value*) untuk beberapa *field* pada sebuah *class*. Konsekuensinya, semua objek yang dibuat dari *class* tersebut akan selalu memiliki nilai *field* yang sama satu sama lainnya. Pada contoh sebelumnya, semua objek yang dibuat dari *class Mobil* akan selalu bermodel “Hatchback” dari pabrikan “Toyota”.

Dengan demikian, Anda tidak memiliki kendali terhadap nilai default untuk setiap *instance* dari *class* tersebut. Sehingga, Anda tidak bisa mengganti nilai-nilai default tersebut sesuai dengan keinginan Anda.

Agar Anda bisa menetapkan nilai *field* yang berbeda-beda untuk setiap objek yang dibuat dari sebuah *class*, Anda bisa menggunakan *constructor* berparameter atau *parameterized constructor*.

Constructor berparameter

Apabila Anda membuat sebuah *class* seperti *class Mobil* pada contoh sebelumnya, Anda tentunya tidak ingin semua objek dari *class* tersebut memiliki nilai *pabrikan* dan *model* yang sama, bukan?

Pastinya, Anda ingin membuat setiap objek mobil yang dibuat dari *class Mobil* tersebut bisa dibedakan antara satu dengan lainnya. Misalnya, Anda menginginkan objek mobil dari pabrikan Toyota, Mazda, Mercedes, Tesla, dan lain-lain. Selain itu, Anda juga menginginkan objek-objek tersebut mempunyai model yang bervariasi seperti hatchback, MPV, SUV, sedan, truck, atau yang lainnya.

Untuk mewujudkannya, Anda bisa menggunakan *constructor* berparameter atau *parameterized constructor*.

Karena *constructor* sejatinya merupakan sebuah *method*, maka Anda pun diperbolehkan mempunyai lebih dari satu parameter untuk sebuah *constructor* pada saat mendefinisikannya.

Kode program berikut ini menggunakan *constructor* berparameter untuk menginisialisasi setiap *field* yang diperlukan.

```
using System;

namespace KodeProgram_4_2
{
    class Mobil
    {
        public string pabrikan;
```

```

    public string model;
    public int thnPembuatan;

    // Parameterized Constructor
    public Mobil(string a, string b, int c)
    {
        pabrik = a;
        model = b;
        thnPembuatan = c;
    }
}

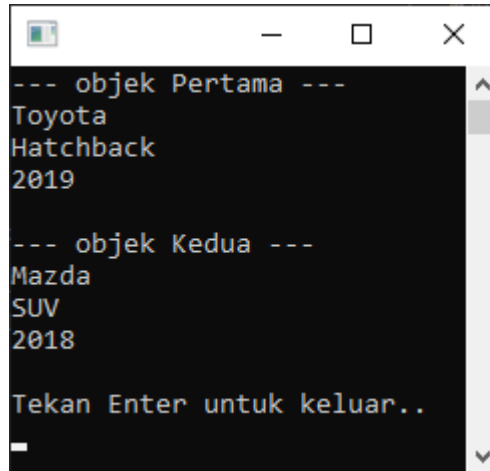
class Program
{
    static void Main(string[] args)
    {
        // objek pertama dari class "Mobil"
        Console.WriteLine("--- objek Pertama ---");
        Mobil mobilToyota = new Mobil("Toyota", "Hatchback", 2019);
        Console.WriteLine(mobilToyota.pabrik);
        Console.WriteLine(mobilToyota.model);
        Console.WriteLine(mobilToyota.thnPembuatan);

        // objek kedua dari class "Mobil"
        Console.WriteLine("\n--- objek Kedua ---");
        Mobil mobilMazda = new Mobil("Mazda", "SUV", 2018);
        Console.WriteLine(mobilMazda.pabrik);
        Console.WriteLine(mobilMazda.model);
        Console.WriteLine(mobilMazda.thnPembuatan);

        Console.WriteLine("\nTekan Enter untuk keluar..");
        Console.ReadLine();
    }
}

```

Kode program 4-2 di atas menghasilkan keluaran seperti yang ditunjukkan pada gambar 4.6.

A screenshot of a terminal window with a black background and white text. The window has standard OS window controls (minimize, maximize, close) at the top. The text inside the terminal is as follows:

```
--- objek Pertama ---  
Toyota  
Hatchback  
2019  
  
--- objek Kedua ---  
Mazda  
SUV  
2018  
  
Tekan Enter untuk keluar..  
_
```

Gambar 4.6

Seperti yang bisa Anda lihat, kali ini kita menggunakan tiga buah parameter, yaitu parameter “a” dan “b” yang masing-masing bertipe data `string`, serta parameter “c” yang bertipe data `int` untuk menetapkan sebuah nilai ke masing-masing *field* dari *class* `Mobil`.

Dengan demikian, kita bebas menentukan dari mana pabrikan, apa jenisnya, dan kapan tahun pembuatan objek yang kita buat. Sehingga, ketika kita membuat dua objek yang berbeda dengan nilai parameter yang berbeda pula, maka masing-masing objek tersebut akan memiliki informasi default yang juga berbeda satu sama lainnya.

Copy Constructor

Ketika Anda ingin membuat lebih dari satu objek, namun dengan karakteristik yang sama persis, Anda bisa melakukannya seperti pada potongan kode program di bawah ini.

```
Mobil mobilToyota1 = new Mobil("Toyota", "Hatchback", 2019);
```

```
Mobil mobilToyota2 = new Mobil("Toyota", "Hatchback", 2019);
```

Baik `mobilToyota1` maupun `mobilToyota2` merupakan mobil keluaran Toyota dengan tipe hatchback yang diproduksi tahun 2019. Namun demikian, keduanya merupakan dua objek yang berbeda.

Namun, saya tidak menyarankan Anda menggunakan cara seperti di atas. Anda bisa menggunakan cara yang lebih baik disamping menggunakan cara di atas, yaitu dengan menggunakan *copy constructor*.

Sebenarnya, C# tidak menyediakan *copy constructor* secara eksplisit. Namun jika diperlukan, seperti pada skenario di atas, Anda bisa membuat sebuah *constructor* yang bisa melakukan tugas seperti sebuah *copy constructor*.

Anda bisa membuat sebuah *copy constructor* dengan cara **menggunakan class itu sendiri sebagai tipe data pada parameter constructor-nya**. Agar Anda bisa memahaminya dengan lebih baik, perhatikan kode program berikut ini.

```
class Mobil
```



```

{
    public string pabrik;
    public string model;
    public int thnPembuatan;

    // Parameterized Constructor
    public Mobil(string a, string b, int c)
    {
        pabrik = a;
        model = b;
        thnPembuatan = c;
    }

    // Copy Constructor
    public Mobil(Mobil mobil)
    {
        pabrik = mobil.pabrik;
        model = mobil.model;
        thnPembuatan = mobil.thnPembuatan;
    }
}

class Program
{
    static void Main(string[] args)
    {
        // objek mobilToyota1
        Console.WriteLine("--- objek mobilToyota1 ---");
        Mobil mobilToyota1 = new Mobil("Toyota", "Hatchback", 2019);
        Console.WriteLine(mobilToyota1.pabrik);
        Console.WriteLine(mobilToyota1.model);
        Console.WriteLine(mobilToyota1.thnPembuatan);

        // objek mobilToyota2
        Console.WriteLine("\n--- objek mobilToyota2 ---");
        // kali ini objek mobilToyota1 digunakan
        // sebagai parameter pada constructor
    }
}

```

```

// sehingga copy constructor akan dipanggil
Mobil mobilToyota2 = new Mobil(mobilToyota1);
Console.WriteLine(mobilToyota2.pabrikan);
Console.WriteLine(mobilToyota2.model);
Console.WriteLine(mobilToyota2.thnPembuatan);

Console.WriteLine("\nTekan Enter untuk keluar..");
Console.ReadLine();
}
}

```

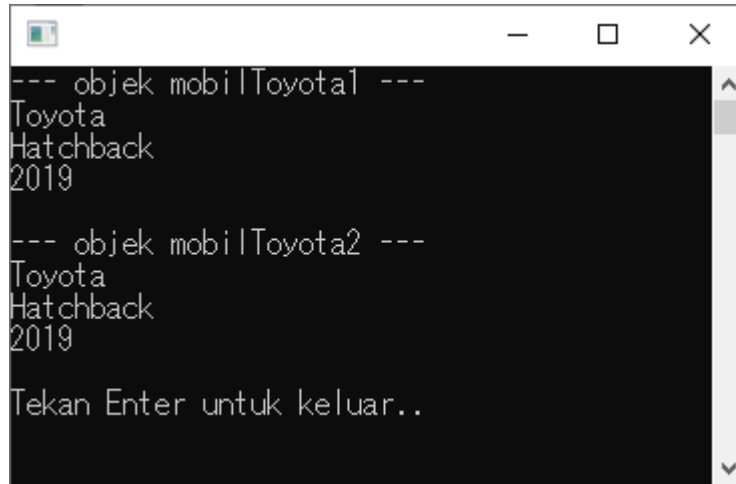
Pada kode program di atas, kita memodifikasi kode program 4-2 dengan menambahkan sebuah *constructor* yang menggunakan tipe *class*-nya sendiri, yaitu tipe `Mobil`, sebagai parameter. *Constructor* tersebut nantinya bisa kita gunakan sebagai *copy constructor* dari *class* `Mobil`.

Kemudian, kita membuat dua objek berbeda, yaitu `mobilToyota1` dan `mobilToyota2`.

Pada saat membuat objek `mobilToyota1`, kita memberikan tiga buah argumen pada *constructor*-nya. Oleh karena itu, *constructor* `public Mobil(string a, string b, int c)` lah yang dipanggil.

Sedangkan pada saat membuat objek `mobilToyota2`, kita memberikan objek `mobilToyota1`, yang sudah diinstansiasi sebelumnya, sebagai argumen untuk *constructor*-nya. Sehingga kali ini, *constructor* `public Mobil(Mobil mobil)` lah yang dipanggil. Di dalam *constructor* ini, karakteristik objek `mobilToyota1` disalin menjadi karakteristik objek `mobilToyota2`. Itulah mengapa *constructor* ini disebut dengan *copy constructor*.

Kode program di atas akan menghasilkan keluaran seperti yang ditampilkan pada gambar 4.7.



```
--- objek mobilToyota1 ---  
Toyota  
Hatchback  
2019  
  
--- objek mobilToyota2 ---  
Toyota  
Hatchback  
2019  
  
Tekan Enter untuk keluar..
```

Gambar 4.7

Kata kunci “this”

Dalam pemrograman C#, kata kunci **this** digunakan untuk mengacu pada objek (*instance*) dari *class* yang dibuat saat ini. Jadi, misalnya Anda membuat objek bernama mobilToyota dari *class* **Mobil**, kata kunci **this** di dalam *class* **Mobil** ini nantinya akan “mewakili” objek mobilToyota yang Anda buat tadi.

Kata kunci **this** ini sangat berguna, misalnya, ketika Anda membuat *constructor* berparameter yang memiliki nama parameter yang sama persis dengan nama *field* dari *class* yang bersangkutan.

Perhatikan kode program berikut ini.

```
using System;
```

```
namespace KodeProgram_4_3
```

```
{  
    class Mobil  
    {  
        public string pabrik;   
        public string model;   
        public int thnPembuatan;  
  
        // Parameterized Constructor  
        public Mobil(string pabrik, string model, int thnPembuatan)  
        {  
            // Gunakan kata kunci this untuk membedakan antara  
            // field dan parameter  
  
            // sebelah kiri operator adalah nama field dari  
            // objek saat ini (this object)  
  
            // sebelah kanan operator adalah nama parameter dari  
            // constructor  
  
            this.pabrik = pabrik;  
            this.model = model;  
            this.thnPembuatan = thnPembuatan;  
        }  
    }  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        // objek pertama dari class "Mobil"  
        Console.WriteLine("--- objek Pertama ---");  
        Mobil mobilToyota = new Mobil("Toyota", "Hatchback", 2019);  
        Console.WriteLine(mobilToyota.pabrik);  
        Console.WriteLine(mobilToyota.model);  
    }  
}
```

```

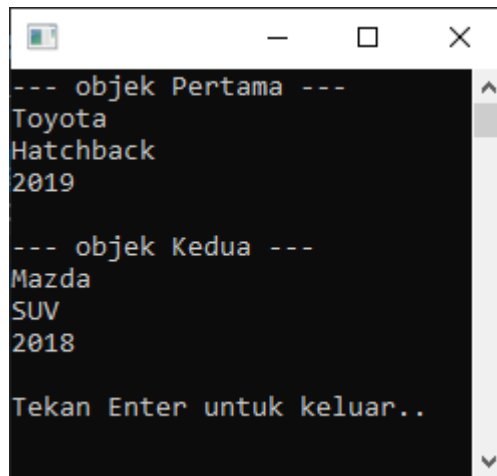
Console.WriteLine(mobilToyota.thnPembuatan);

// objek kedua dari class "Mobil"
Console.WriteLine("\n--- objek Kedua ---");
Mobil mobilMazda = new Mobil("Mazda", "SUV", 2018);
Console.WriteLine(mobilMazda.pabrikan);
Console.WriteLine(mobilMazda.model);
Console.WriteLine(mobilMazda.thnPembuatan);

Console.WriteLine("\nTekan Enter untuk keluar..");
Console.ReadLine();
    }
}
}

```

Ketika Anda menjalankan kode program di atas, Anda akan mendapatkan hasil seperti yang ditampilkan pada Gambar 4.8.



Gambar 4.8

Jika Anda perhatikan, hasilnya sama persis seperti yang ditampilkan pada Gambar 4.3. Hanya saja, apabila Anda menggunakan kata kunci **this** untuk

mengacu pada objek yang bersangkutan, maka nama parameter yang Anda sediakan bisa menjadi lebih masuk akal ketimbang nama-nama seperti “a”, “b”, atau “c”.

Karena kata kunci **this** ini mengacu pada objek yang dibuat saat ini, maka Anda juga bisa memberikan kata kunci **this** ini sebagai argumen untuk sebuah *method*.

Untuk lebih memahaminya, perhatikan kode program berikut ini.

```
using System;

namespace KodeProgram_4_4
{
    class Mobil
    {
        public string pabrikan;
        public string model;
        public int thnPembuatan;
        public int kondisi;

        public Mobil(string pabrikan, string model, int thnPembuatan, int
kondisi)
        {
            this.pabrikan = pabrikan;
            this.model = model;
            this.thnPembuatan = thnPembuatan;
            this.kondisi = kondisi;
        }

        public void getInfoMobil()
        {
            Console.WriteLine("Pabrikan: {0}", pabrikan);
            Console.WriteLine("Tipe Kendaraan: {0}", model);
        }
    }
}
```

```

        Console.WriteLine("Tahun Pembuatan: {0}", thnPembuatan);

        // memberikan objek saat ini
        // sebagai argumen ke sebuah method
        Console.WriteLine("Kelayakan: {0}",
UjiKelayakan.CekKelayakan(this));
    }
}

class UjiKelayakan
{
    public static string kelayakan;

    public static string CekKelayakan(Mobil mobil)
    {
        // Salah satu field dari objek saat ini
        // digunakan untuk perbandingan nilai
        if (mobil.kondisi < 60)
        {
            kelayakan = "Tidak Layak Jalan!";
        }
        else
        {
            kelayakan = "Masih Layak Jalan!";
        }

        return kelayakan;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Mobil mobilToyota = new Mobil("Toyota", "Hatchback", 2019, 90);
        mobilToyota.getInfoMobil();
    }
}

```

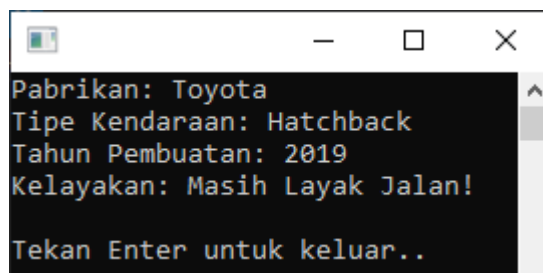
```
        Console.WriteLine("\nTekan Enter untuk keluar..");  
        Console.ReadLine();  
    }  
}  
}
```

Pada kode program 4-4 di atas, kita memberikan objek `mobilToyota` yang kita buat saat ini sebagai sebuah argumen untuk *method* `CekKelayakan(Mobil mobil)` melalui kata kunci **this**.

```
UjiKelayakan.CekKelayakan(this); // objek mobilToyota “diwakili” oleh this
```

Method ini berfungsi untuk menguji apakah kondisi objek `mobilToyota` ini masih layak jalan atau tidak. Apabila kondisi dari objek `mobilToyota` ini kurang dari 60 maka *method* ini akan mengembalikan nilai bertipe `string` yang berisi **“Tidak layak jalan!”**. Sebaliknya, jika kondisinya lebih atau sama dengan 60, maka *method* ini akan mengembalikan nilai yang berisi **“Masih layak jalan!”**.

Ketika Anda menjalankan kode program 4-4 di atas, Anda akan mendapatkan hasil seperti yang ditampilkan pada Gambar 4.9.



Gambar 4.9

Menggunakan lebih dari satu constructor dalam satu class

Bayangkan Anda sedang berencana untuk membangun sebuah rumah (objek). Katakanlah Anda memiliki dua pilihan kontraktor (*constructor*) untuk mengerjakan rancang bangun (*class*) rumah Anda, yaitu Pakde Owi dan Om Owo.

Pakde Owi adalah seorang kontraktor yang lebih memilih untuk menggunakan material alami seperti bata merah untuk dindingnya, kayu sengon untuk rangka atapnya, genteng tanah liat untuk atapnya, dan lain sebagainya.

Sedangkan Om Owo adalah seorang kontraktor kekinian yang lebih memilih material dengan teknologi terkini seperti bata hebel, rangka baja, genteng keramik, dan lain-lain.

Kontraktor yang Anda pilih akan menentukan ***bagaimana*** rumah (objek) Anda dibangun nantinya. Namun demikian, siapapun kontraktor yang Anda pilih, Anda akan mendapatkan sebuah rumah (objek) yang sesuai dengan rancang bangun (*class*) yang telah Anda sediakan. Yang berbeda hanyalah material bangunannya saja (kondisi atau karakteristik objeknya).

Dengan konsep seperti ini, terkadang Anda juga akan membutuhkan lebih dari satu pilihan *constructor* pada *class* yang Anda buat.

Perhatikan potongan kode program di bawah ini untuk kembali bermain dengan *class Mobil* seperti pada beberapa kode program sebelumnya.

```
class Mobil
{
    public string pabrik;
    public string model;
    public int thnPembuatan;

    // Constructor pertama
    public Mobil(string pabrik, string model, int thnPembuatan)
    {
        this.pabrik = pabrik;
        this.model = model;
        this.thnPembuatan = thnPembuatan;
    }

    // Constructor kedua
    public Mobil(string pabrik, int thnPembuatan)
    {
        this.pabrik = pabrik;
        this.thnPembuatan = thnPembuatan;
    }
}
```

Di dalam `class Mobil` pada contoh kode program di atas, kita menyediakan dua buah *constructor* untuk membuat sebuah objek dari *class* tersebut.

Constructor pertama menginisialisasi semua *field* yang tersedia di dalam `class Mobil`, sementara *constructor* kedua hanya menginisialisasi *field* bernama `pabrik` dan `thnPembuatan`.

Pada saat membuat sebuah aplikasi, terkadang Anda memerlukan beberapa objek dari sebuah *class* yang sama dengan kondisi awal yang berbeda-beda. Dengan menyediakan beberapa pilihan *constructor* di dalam *class* tersebut,

hal ini akan mudah Anda dapatkan.

Namun Anda mungkin berpikir, *constructor* mana yang akan dieksekusi ketika Anda membuat sebuah objek baru dari *class* tersebut. Jawabannya adalah dengan menggunakan konsep *constructor overloading* yang akan saya jelaskan selanjutnya.

Constructor overloading

Karena *constructor* sejatinya merupakan sebuah *method*, maka Anda juga bisa melakukan *overloading* terhadap *constructor*. Caranya adalah dengan membuat lebih dari satu *constructor* dengan nama *method* yang sama tetapi dengan parameter yang berbeda-beda.

Contoh kode program berikut ini mengimplementasikan konsep *constructor overloading* dalam Bahasa pemrograman C#.

```
using System;

namespace KodeProgram_4_5
{
    class Mobil
    {
        public string pabrik;
        public string model;
        public int thnPembuatan;

        // Constructor pertama tanpa parameter
        public Mobil()
        {
            pabrik = "Toyota";
            model = "Hatchback";
        }
    }
}
```

```

        thnPembuatan = 2019;
    }

    // Constructor kedua dengan parameter
    public Mobil(string pabrik, string model, int thnPembuatan)
    {
        this.pabrik = pabrik;
        this.model = model;
        this.thnPembuatan = thnPembuatan;
    }
}

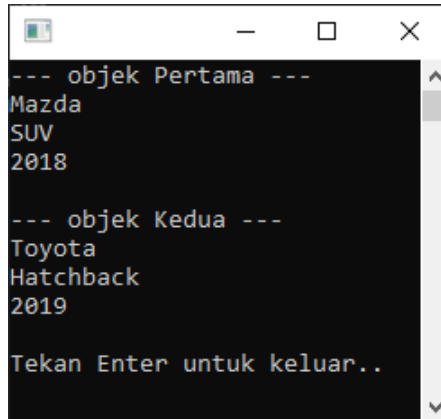
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("--- objek Pertama ---");
        Mobil mobilMazda = new Mobil("Mazda", "SUV", 2018);
        Console.WriteLine(mobilMazda.pabrik);
        Console.WriteLine(mobilMazda.model);
        Console.WriteLine(mobilMazda.thnPembuatan);

        Console.WriteLine("\n--- objek Kedua ---");
        Mobil mobil = new Mobil();
        Console.WriteLine(mobil.pabrik);
        Console.WriteLine(mobil.model);
        Console.WriteLine(mobil.thnPembuatan);

        Console.WriteLine("\nTekan Enter untuk keluar..");
        Console.ReadLine();
    }
}

```

Ketika Anda menjalankan kode program 4-5 di atas, Anda akan mendapatkan hasil seperti yang ditampilkan pada Gambar 4.10.



```

--- objek Pertama ---
Mazda
SUV
2018

--- objek Kedua ---
Toyota
Hatchback
2019

Tekan Enter untuk keluar..

```

Gambar 4.10

Pada kode program di atas, pada saat kita membuat objek bernama `mobilMazda`, *constructor* yang dipanggil adalah *constructor* yang kedua. Hal itu dikarenakan pada saat menginstansiasi objek tersebut, kita memberikan argumen yang sesuai dengan parameter yang dimiliki oleh *constructor* kedua.

Sedangkan ketika kita membuat objek bernama `mobil`, *constructor* yang dipanggil adalah *constructor* pertama, yaitu *constructor* yang tidak memiliki parameter. Hal itu dikarenakan kita menginstansiasi objek `mobil` tersebut tanpa memberikan argumen apapun.

Constructor chaining

Anggap saja Anda memiliki perusahaan pabrik mobil imajiner. Pabrik Anda ini pada mulanya hanya memproduksi mobil sedan yang dijual dengan harga Rp. 200,000,000. Anda kemudian membuat model untuk lini produksi mobil Anda tersebut. Apabila dimodelkan dengan sebuah *class*, Anda bisa menuliskannya seperti ini.

```
using System;

namespace PabrikMobil
{
    class Mobil
    {
        public string model;
        public string hargaIDR;

        public Mobil()
        {
            model = "Sedan";
            hargaIDR = "Rp. 200, 000, 000";
        }
    }
}
```

Sukses besar! Perusahaan Anda berhasil mencatatkan penjualan yang luar biasa untuk model mobil ini.

Setelah beberapa lama, Anda memutuskan untuk memproduksi mobil dengan berbagai model berbeda, namun dengan harga yang sama. Masalahnya, lini produksi pabrik Anda sudah didesain hanya untuk memproduksi mobil sedan.

Akhirnya Anda memutuskan untuk membuat lini produksi baru, dan memodelkannya dengan sebuah *class* seperti ini.

```
using System;

namespace PabrikMobil
{
    class Mobil
```

```

{
    public string model;
    public string hargaIDR;

    public Mobil()
    {
        model = "Sedan";
        hargaIDR = "Rp. 200,000,000";
    }

    public Mobil(string model)
    {
        this.model = model;
        hargaIDR = "Rp. 200,000,000";
    }
}
}

```

Lagi, setelah beberapa lama, Anda menyadari bahwa beberapa model mobil yang Anda produksi tidak bisa lagi Anda hargai dengan harga yang sama. Akhirnya, lagi-lagi Anda memutuskan untuk memodelkan sebuah lini produksi baru yang bisa memproduksi berbagai model mobil dengan harga yang bervariasi dengan sebuah *class* seperti ini.

```

using System;

namespace PabrikMobil
{
    class Mobil
    {
        public string model;
        public string hargaIDR;

        public Mobil()

```

```
{
    model = "Sedan";
    hargaIDR = "Rp. 200, 000, 000";
}

public Mobil(string model)
{
    this.model = model;
    hargaIDR = "Rp. 200, 000, 000";
}

public Mobil(string model, int hargaIDR)
{
    this.model = model;
    this.hargaIDR = hargaIDR;
}
}
```

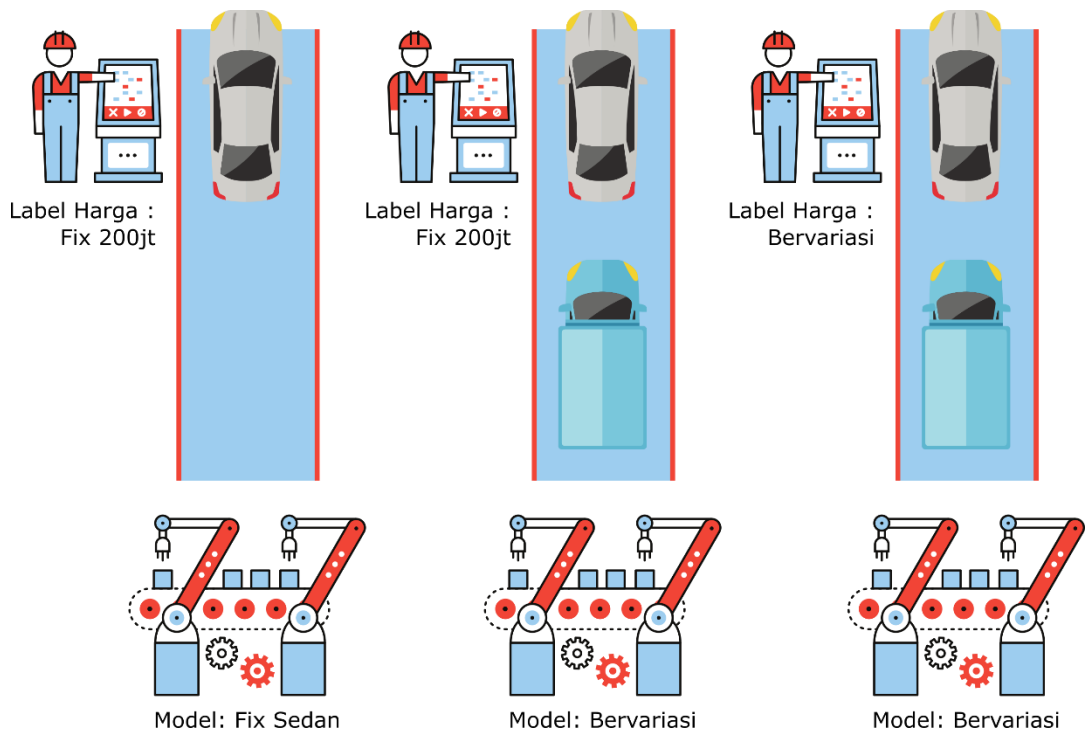
Pada akhirnya, pabrik Anda kini memiliki tiga buah lini produksi untuk memproduksi mobil.

Semua berjalan dengan lancar. Sekarang, pabrik Anda mampu memproduksi segala tipe mobil dengan harga yang bervariasi. Namun kemudian, Anda merasa bahwa model lini produksi pabrik Anda ini kurang efisien. Anda ingin mengurangi jumlah mesin dan operator tanpa harus merubah output yang sudah dihasilkan.

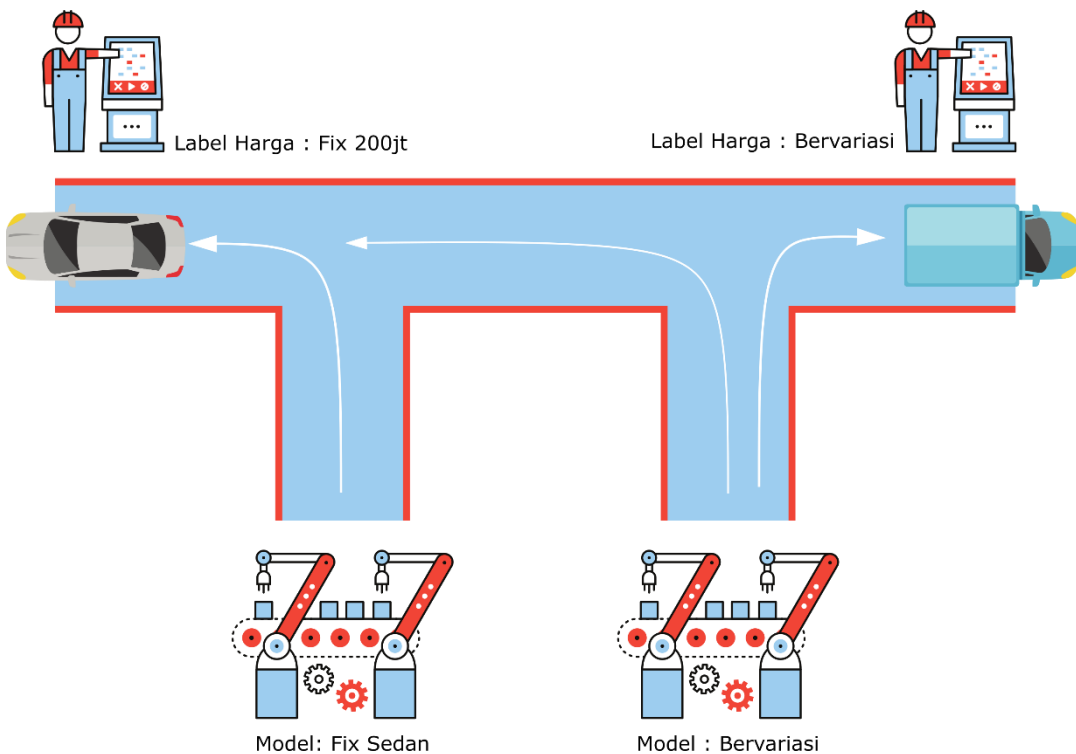
Oleh karena itu, Anda berpikir ulang bagaimana membuat model lini produksi yang terintegrasi. Kalau sebelumnya tiga lini produksi yang Anda miliki beroperasi sendiri-sendiri, kali ini Anda merumuskan sebuah model

yang membuat tiga lini produksi tersebut terkait satu sama lainnya. Dalam kata lain, Anda membuat model *chaining* untuk tiga lini produksi Anda.

Apabila ketiga lini produksi Anda diilustrasikan dalam sebuah sketsa, tiga lini produksi Anda sebelumnya beroperasi seperti pada gambar 4.11, sementara gambar 4.12 mengilustrasikan operasi pada tiga lini produksi pabrik Anda setelah dilakukan *chaining*.



Gambar 4.11



Gambar 4.12

Lihat bagaimana Anda bisa mengurangi jumlah mesin Anda dari 6 mesin menjadi 4 mesin saja.

Untuk memodelkan ilustrasi di Gambar 4.12 dalam bentuk *class*, Anda bisa menggunakan sebuah metode yang bernama *constructor chaining* seperti pada kode program berikut ini.

```
using System;

namespace PabrikMobil
{
    class Mobil
    {
```

```

public string model;
public string hargaIDR;

public Mobil() : this("Sedan")
{
}

public Mobil(string tipe) : this(tipe, "Rp. 200,000,000")
{
}

public Mobil(string model, string hargaIDR)
{
    this.model = model;
    this.hargaIDR = hargaIDR;
}
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("--- objek Mobil Pertama ---");
        Mobil mobil1 = new Mobil();
        Console.WriteLine("Tipe : {0}", mobil1.model);
        Console.WriteLine("Harga Jual: {0}", mobil1.hargaIDR);

        Console.WriteLine("\n--- objek Mobil Kedua ---");
        Mobil mobil2 = new Mobil("Hatchback");
        Console.WriteLine("Tipe : {0}", mobil2.model);
        Console.WriteLine("Harga Jual: {0}", mobil2.hargaIDR);

        Console.WriteLine("\n--- objek Mobil Ketiga ---");
        Mobil mobil3 = new Mobil("SUV", "Rp. 300,000,000");
        Console.WriteLine("Tipe : {0}", mobil3.model);
    }
}

```

```
        Console.WriteLine("Harga Jual: {0}", mobil3.hargaIDR);

        Console.WriteLine("\nTekan Enter untuk keluar..");
        Console.ReadLine();
    }
}
}
```

Berbeda dengan *class* yang Anda modelkan sebelumnya, dengan metode *constructor chaining*, Anda bisa menghindari duplikasi kode di dalam sebuah *constructor* dengan *constructor* yang lain.

Seperti yang bisa Anda lihat, Anda hanya perlu sekali saja menuliskan kode di bawah ini.

```
this.model = model;
this.hargaIDR = hargaIDR;
```

Pada saat Anda membuat (menginstansiasi) objek `mobil1`, *constructor* `Mobil()` akan dipanggil. Dengan menambahkan `this("Sedan")` setelah *constructor* tersebut, kode tersebut akan memanggil *constructor* dengan jumlah parameter yang sama, yaitu *constructor* `Mobil(string model)`.

Lalu, `this(model, "Rp.200,000,000")` akan memanggil *constructor* dengan jenis parameter yang sama, yaitu *constructor* `Mobil(string model, string hargaIDR)`.

Kemudian pada akhirnya nilai untuk `this.model` dan `this.hargaIDR` ditetapkan di dalam *constructor* ini.

Pada saat Anda menginstansiasi objek `mobil2`, Anda melewati satu buah argumen bertipe data `string`. Maka, *constructor* `Mobil(string model)` akan dipanggil.

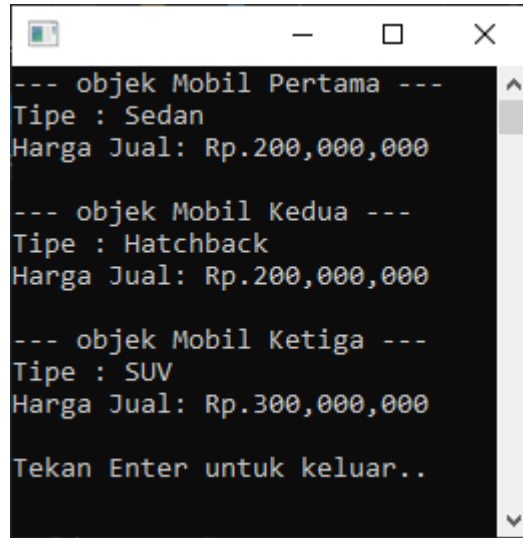
Lalu, kode `this(model, "Rp.200,000,000")` ini akan memanggil *constructor* dengan jumlah parameter yang sama, yaitu *constructor* `Mobil(string model, string hargaIDR)`.

Kemudian pada akhirnya nilai untuk `this.model` dan `this.hargaIDR` ditetapkan di dalam *constructor* ini.

Tentu saja saat menginstansiasi objek `mobil3` dengan dua buah argumen, *constructor* yang akan dipanggil adalah *constructor* dengan dua buah parameter, yaitu `Mobil(string model, string hargaIDR)`. Karena *constructor* ini adalah ujung dari untaian *constructor* yang Anda miliki, maka nilai untuk `this.model` dan `this.hargaIDR` akan segera ditetapkan.

Jadi, *constructor chaining* ini adalah metode untuk membuat untaian antar *constructor*, agar satu *constructor* dapat memanggil *constructor* yang lainnya. Dengan menggunakan *constructor chaining*, Anda dapat menghindari penulisan kode program yang sama secara berulang-ulang (duplikasi kode) ketika Anda menemui skenario seperti pada pabrik mobil imajiner Anda tadi.

Ketika Anda menjalankan kode program di atas, Anda akan mendapatkan hasil seperti yang ditampilkan pada Gambar 4.14.

A screenshot of a Windows-style console application window. The window has a title bar with a small icon on the left and standard minimize, maximize, and close buttons on the right. The console area has a black background with white text. The text is as follows:
--- objek Mobil Pertama ---
Tipe : Sedan
Harga Jual: Rp.200,000,000

--- objek Mobil Kedua ---
Tipe : Hatchback
Harga Jual: Rp.200,000,000

--- objek Mobil Ketiga ---
Tipe : SUV
Harga Jual: Rp.300,000,000

Tekan Enter untuk keluar..
A vertical scrollbar is visible on the right side of the console window, with the text currently at the top.

Gambar 4.14

Bab 5

Mengimplementasikan Enkapsulasi Dengan Sebuah Class

Sering dianggap sebagai pilar pertama pemrograman berorientasi objek, enkapsulasi dapat digunakan untuk menggambarkan aksesibilitas pada setiap anggota (*field*, *method*, dll) sebuah *class*.

Menentukan aksesibilitas anggota penyusun *class* nantinya akan sangat membantu Anda mewujudkan **abstraksi** yang sudah Anda tentukan terlebih dulu pada tahap desain. Oleh karena itu, dapat mengimplementasikan enkapsulasi dengan baik akan membantu Anda mendapatkan abstraksi yang baik pula.

C# sendiri menyediakan *access modifier* dan properti untuk membantu Anda mengimplementasikan enkapsulasi di dalam sebuah *class*.

Mengenal istilah *assembly* di lingkungan .NET

Sebelum Anda mempelajari teknik-teknik untuk mengimplementasikan enkapsulasi pada sebuah *class*, ada baiknya Anda terlebih dahulu memahami

konstruksi sebuah program komputer di platform .NET.

Dalam pemrograman C#, sekedar menulis kode program saja tidak cukup membuat program Anda bisa berfungsi sebagai mestinya. Hal ini dikarenakan C# adalah bahasa pemrograman yang perlu dikompilasi terlebih dahulu agar bisa dijalankan.

Pada saat mengkompilasi kode program yang sudah Anda tulis, *compiler* akan membaca setiap baris kode program tersebut untuk kemudian mengubahnya menjadi sebuah file yang dapat dijalankan. Dalam lingkungan .NET, Hasil dari kompilasi ini disebut dengan *assembly*. *Assembly* sendiri terdiri dari dua jenis, yaitu *process assembly (executable file/EXE)* dan *library assembly (dynamic link library/DLL)*.

Anda pastinya sudah sangat familiar dengan file EXE, bukan? Betul, sebuah file dengan ekstensi .exe biasanya adalah file yang Anda pilih untuk menjalankan sebuah aplikasi di komputer Anda.

Namun, bagaimana dengan DLL? Mungkin Anda sudah beberapa kali melihat file-file berekstensi .dll di dalam folder komputer Anda. Namun, Anda mungkin masih penasaran apa fungsi dari file-file DLL tersebut dan mengapa file-file tersebut ada di dalam folder komputer Anda?

Masih ingat buku alamat yang kita bahas di Bab 1?

Agar fitur buku alamat ini bisa Anda gunakan kembali untuk setiap aplikasi software yang membutuhkan fitur tersebut tanpa harus menulis dan

mengkompilasi ulang kode programnya, Anda perlu menuliskan kode program buku alamat tersebut pada sebuah *project* yang berbeda dan kemudian mengkompilasinya sebagai sebuah komponen *software*.

Komponen *software* ini dikompilasi menjadi sebuah file DLL yang nantinya bisa dijadikan sebagai referensi ketika Anda membutuhkan sebuah komponen atau fitur tertentu. Jadi, ketika Anda, katakanlah, ingin membuat lebih dari satu aplikasi berbeda yang masing-masing membutuhkan fitur buku alamat, daripada setiap kali harus menuliskan kode program yang sama untuk masing-masing aplikasi tersebut, Anda cukup membuat masing-masing aplikasi tersebut mereferensi atau mengacu pada sebuah file DLL yang memuat fungsi-fungsi buku alamat tadi.

Pada saat kode program yang Anda tulis perlu mereferensi ke sebuah file DLL, artinya aplikasi Anda (yang juga merupakan sebuah *assembly*) membutuhkan *assembly* lain (file DLL tadi) agar dapat dijalankan.

Itulah mengapa ketika Anda mencoba menghapus sebuah file DLL dari komputer Anda, jangan kaget apabila ada aplikasi yang kemudian tidak bisa dijalankan karena kehilangan referensi.

Mengakses anggota class

Dalam pemrograman C#, *access modifiers* adalah kata kunci yang digunakan untuk menentukan tingkat aksesibilitas untuk setiap anggota penyusun *class* (*class member*) dari *assembly* yang sama maupun dari *assembly* yang

berbeda.

Di bagian ini, Anda akan mempelajari beberapa *access modifiers* yang dapat diterapkan pada anggota penyusun *class* untuk mengontrol bagaimana mereka dapat diakses oleh kode lain dalam sebuah *assembly*.

Menggunakan *access modifier* untuk setiap anggota penyusun *class* adalah salah satu cara melakukan enkapsulasi yang memungkinkan Anda untuk membatasi akses terhadap setiap anggota penyusun *class*.

public

Access modifier **public** digunakan untuk menyatakan bahwa sebuah anggota penyusun *class* tidak memiliki batasan akses. Artinya, mereka dapat diakses oleh kode program dari *assembly* manapun selama *assembly* tersebut mereferensi *assembly* di mana anggota penyusun *class* tersebut dimuat.

private

Dengan mendeklarasikan sebuah anggota penyusun *class* sebagai **private**, artinya Anda membatasi akses terhadap anggota penyusun *class* tersebut hanya dari dalam *class* di mana mereka dideklarasikan. Anda tidak mengizinkan anggota penyusun *class* tersebut bisa diakses dari *class* yang lain meskipun *class* tersebut masih berada dalam satu *assembly*.

Catatan: Apabila Anda tidak memberikan *access modifier* untuk sebuah

anggota penyusun *class* secara eksplisit, secara default, anggota penyusun *class* tersebut akan memiliki batasan akses yang sama seperti ketika Anda mendeklarasikannya sebagai **private**.

protected

Untuk membatasi akses terhadap anggota penyusun *class* hanya dari dalam *class* itu sendiri dan juga *class* lain yang mewarisi *class* tersebut, Anda bisa menggunakan kata kunci **protected** pada anggota penyusun *class* tersebut.

Kata kunci **protected** ini akan sangat berguna ketika Anda ingin membatasi akses ke sebuah anggota penyusun *class* saat mengimplementasikan konsep pewarisan pada sebuah *class*.

internal

Katakanlah saya membuat sebuah *class* untuk melakukan kalkulasi aritmatika seperti penjumlahan, pengurangan, pembagian, dan perkalian. *Class* ini saya kompilasi sebagai sebuah komponen software dan menghasilkan sebuah file DLL yang bisa Anda gunakan sebagai referensi untuk melakukan kalkulasi di dalam aplikasi Anda.

Di dalam komponen tersebut, saya mendeklarasikan semua anggota penyusun *class* dengan kata kunci **internal**. Kira-kira apa yang terjadi?

Meskipun Anda sudah melakukan referensi ke komponen *software* yang saya buat, Anda sama sekali tidak bisa mengakses fungsi-fungsi yang

terdapat pada *class* yang berada di dalam komponen tersebut. Maaf, ternyata saya melakukan kesalahan.

Dengan menggunakan kata kunci **internal** untuk sebuah anggota penyusun *class*, artinya mereka hanya bisa diakses oleh kode program yang masih berada dalam satu *assembly*.

Komponen software yang saya buat dan aplikasi yang Anda buat adalah dua *assembly* yang berbeda. Oleh karena itu, apabila saya menggunakan kata kunci **internal** untuk setiap anggota penyusun *class* pada komponen software tadi, kode program di aplikasi Anda tidak dapat mengakses anggota penyusun *class* yang berada di dalam komponen software tersebut.

Properti

Salah satu cara untuk menerapkan enkapsulasi adalah dengan cara mendeklarasikan *field* dari sebuah *class* sebagai **private**. Cara ini biasa disebut dengan penyembunyian data atau *data hiding*. Artinya, data tersebut dilindungi dari akses dan modifikasi langsung oleh kode program yang tidak memiliki akses langsung terhadap data tersebut.

Namun demikian, Data tersebut masih bisa Anda buka kepada pengguna *class* Anda dengan cara menyediakan “*antarmuka*” yang berfungsi untuk mendapatkan atau menentukan sebuah nilai pada sebuah *field* di dalam *class* Anda tersebut.

Antarmuka tersebut bisa berupa dua buah *method* yang masing-masing memiliki akses **public**. Fungsi dari *method* pertama adalah untuk menetapkan sebuah nilai ke dalam sebuah *field*, sementara *method* kedua berfungsi untuk mendapatkan nilai dari sebuah *field*.

Coba perhatikan kode program 5-1 berikut ini.

```
using System;

namespace KodeProgram_5_1
{
    class SebuahClass
    {
        // data field
        private int x;

        // Method untuk menentukan nilai x
        public void SetX(int i)
        {
            x = i;
        }

        // Method untuk mendapatkan nilai x
        public int GetX()
        {
            return x;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            SebuahClass sc = new SebuahClass();
        }
    }
}
```

```

        // Menetapkan nilai X dilakukan
        // melalui sebuah "antarmuka"
        // method SetX()
        sc.SetX(10);

        // Mendapatkan nilai X juga dilakukan
        // melalui sebuah "antarmuka"
        // method GetX()
        int xVal = sc.GetX();

        Console.WriteLine(xVal);
    }
}

```

Apabila Anda perhatikan, untuk mendapatkan nilai atau menetapkan nilai ke dalam *field* `x`, Anda dapat menggunakan dua buah *method* bernama `SetX(int i)` dan `GetX()` yang disediakan oleh *class* `SebuahClass`.

Sampai di sini mungkin Anda bertanya, jika pada akhirnya *field* yang dideklarasikan sebagai **private** dapat diakses dari luar, lalu mengapa *field* tersebut dari awal harus dideklarasikan sebagai **private**?

Jawabannya, dengan menggunakan “antarmuka” sebuah *method*, salah satunya, Anda bisa mengimplementasikan sebuah verifikasi untuk memvalidasi sebuah nilai input. Apakah input tersebut valid atau tidak. Dengan mendeklarasikan *field* sebagai **public**, maka Anda tidak bisa memvalidasi nilai input yang akan merubah nilai dari *field* tersebut.

Mari kita lakukan sedikit modifikasi pada *method* `SetX(int i)` di kode

program 5-1.

```
using System;

namespace KodeProgram_5_2
{
    class SebuahClass
    {
        // data field
        private int x;

        // Method untuk menentukan nilai x
        public void SetX(int i)
        {
            // Verifikasi nilai i
            if (i >= 10)
            {
                x = i;
            }
        }

        // Method untuk mendapatkan nilai x
        public int GetX()
        {
            return x;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            SebuahClass sc = new SebuahClass();

            // Menetapkan nilai X dilakukan
            // melalui sebuah "antarmuka"
```

```

        // method SetX()
        sc.SetX(10);

        // Mendapatkan nilai X juga dilakukan
        // melalui sebuah "antarmuka"
        // method GetX()
        int xVal = sc.GetX();

        Console.WriteLine(xVal);
    }
}

```

Pada kode program 5-2 di atas, di dalam *method* `SetX(int i)`, kita memberikan kondisi untuk memvalidasi nilai `i`. Kita hanya mengizinkan nilai `i` dirubah apabila `i` bernilai 10 atau lebih.

Untungnya dalam pemrograman C#, Anda tidak perlu membuat *method* sendiri untuk menyediakan "antarmuka" terhadap sebuah *field*. C# memiliki mekanisme bawaan yang disebut properti untuk melakukan sesuatu seperti pada contoh sebelumnya.

Anda bisa mendeklarasikan sebuah properti dengan cara seperti berikut ini.

```

<access_modifier> <return_type> <property_name>
{
    get
    {
        // untuk mendapatkan nilai
    }

    set
    {

```



```

        // untuk menentukan nilai
    }
}

```

Dimana `<access_modifier>` dapat berupa **public**, **private**, **protected**, atau **internal**. Sedangkan `<return_type>` dapat berupa tipe data C# apapun yang valid. Lalu, `get` dan `set` disebut dengan **aksesor**.

Dengan demikian, kode program 5-2 bisa kita modifikasi sebagai berikut.

```

using System;

namespace KodeProgram_5_3
{
    class SebuahClass
    {
        // data field sebagai private
        private int x;

        // property sebagai public
        public int X
        {
            get
            {
                return x;
            }
            set
            {
                if (value >= 10)
                {
                    x = value;
                }
            }
        }
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        SebuahClass mc = new SebuahClass();
        // Menetapkan nilai x melalui properti
        mc.X = 10;

        // Mendapatkan nilai x melalui properti
        int xVal = mc.X;

        Console.WriteLine(xVal);
    }
}

```

Di dalam aksesor **get**, kita menggunakan kata kunci **return** untuk mendapatkan kembali nilai dari *field* **x**. Sedangkan di dalam aksesor **set**, kita menggunakan kata kunci **value** untuk menentukan nilai yang akan ditetapkan ke dalam variabel **x**.

Tipe-tipe properti

Anda bisa membuat dua jenis properti dalam sebuah *class*, yaitu *read-only property* dan *read-write property*. Meski demikian, Anda sebenarnya bisa juga membuat *write-only property* meskipun ini tidak lazim dilakukan.

Seperti yang sudah Anda pahami, aksesor **get** digunakan untuk mendapatkan nilai dari sebuah *field*, sedangkan aksesor **set** digunakan untuk menetapkan nilai ke sebuah *field*.

Pada contoh sebelumnya, kita mendeklarasikan sebuah properti dengan kedua *accessor* tersebut supaya bisa digunakan baik untuk mendapatkan nilai atau menentukan nilai dari/ke sebuah *field*. Dengan demikian, properti ini disebut bertipe *read-write*.

Sementara, agar sebuah properti hanya bisa digunakan untuk mendapatkan nilai dari sebuah *field* (*read only*), maka Anda tidak perlu mencantumkan *accessor* **set** di properti yang Anda deklarasikan.

Hindari mendeklarasikan sebuah field sebagai public

Untuk mendapatkan enkapsulasi yang baik, mendeklarasikan sebuah *field* sebagai **public** adalah ide yang buruk. Mengapa?

Coba bayangkan sebuah skenario ketika Anda mendeklarasikan sebuah *field* sebagai **public** seperti pada contoh program di bawah ini.

```
using System;

namespace KodeProgram_5_4
{
    class SebuahClass
    {
        public int bilBulat;
    }

    class Program
    {
        static void Main(string[] args)
        {
            SebuahClass obyek = new SebuahClass();
            obyek.bilBulat = 100;
        }
    }
}
```

```

        int hasilBagi = 1000 / obyek.bilBulat;
        Console.WriteLine(hasilBagi);

        Console.ReadLine();
    }
}

```

Sekilas kode program di atas nampaknya tidak ada masalah. Namun di kemudian hari, Anda merasa bahwa untuk menetapkan nilai ke dalam *field* `bilBulat`, diperlukan sebuah persyaratan supaya tidak semua nilai bisa ditetapkan ke dalam *field* tersebut.

Misalnya, Anda ingin mencegah angka 0 ditetapkan ke dalam *field* tersebut agar tidak terjadi pembagian oleh angka 0 yang bisa menyebabkan aplikasi Anda mengalami *crash*.

Dengan ide seperti ini, Anda kemudian memutuskan untuk mengekspos `bilBulat` melalui sebuah properti dan merubah aksesibilitas terhadap *field* tersebut dari **public** menjadi **private**. Dengan demikian, kode program Anda menjadi seperti berikut ini.

```

using System;

namespace KodeProgram_5_5
{
    class SebuahClass
    {
        private int bilBulat;
        public int BilanganBulat
        {

```

```

        get
        {
            return bilBulat;
        }

        set
        {
            if (value != 0)
            {
                bilBulat = value;
            }
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        SebuahClass obyek = new SebuahClass();
        obyek.BilanganBulat = 100;
        int hasilBagi = 1000 / obyek.BilanganBulat;
        Console.WriteLine(hasilBagi);

        Console.ReadLine();
    }
}

```

Sekilas sampai di sini, skenario seperti ini memang tampak tidak ada masalah. Anda hanya perlu merubah implementasi dari *class* `SebuahClass`, lalu melakukan penyesuaian terhadap kode program Anda di mana *class* tersebut digunakan. Pada contoh di atas, Anda hanya perlu merubah kode program `obyek.bilBulat` menjadi `obyek.BilanganBulat`.

Tapi, bayangkan jika Anda bekerja di dalam sebuah tim. Tugas Anda hanyalah mendesain dan membuat sebuah *class* yang nantinya akan didistribusikan ke anggota tim yang lain untuk digunakan.

Awalnya Anda mendesain *class* tersebut seperti pada contoh kode program 5-4 di mana *field* `bilBulat` dideklarasikan sebagai **public**. Kemudian, *class* tersebut anda *compile* menjadi file DLL yang siap didistribusikan. Ketika anggota tim yang lain menggunakannya sebagai referensi, mereka tidak menemukan adanya masalah. Tim Anda gembira, Anda pun gembira.

Namun, seiring berjalannya waktu, akhirnya ditemukan sebuah *bug* pada aplikasi yang dibuat oleh tim Anda. Pada saat pengguna aplikasi tersebut memasukkan angka 0, terjadi eksepsi. Aplikasi tersebut *crash* karena terjadi pembagian oleh angka 0. Tidak!

Setelah mendapat laporan seperti ini, Anda kemudian bergegas mendesain ulang *class* tersebut. Anda memutuskan untuk mengekspos *field* `bilBulat` melalui sebuah properti dan merubah aksesibilitas *field* tersebut dari **public** menjadi **private**.

Anda kemudian kembali mendistribusikan *class* yang sudah Anda modifikasi tersebut. Tidak butuh waktu lama, Anda sudah mendapat protes keras dari anggota tim Anda karena kini mereka tidak lagi bisa mengakses *field* `bilBulat`. Solusinya, anggota tim Anda mau tidak mau harus merubah kode program mereka agar sesuai dengan implementasi *class* Anda yang baru.

Tentunya kali ini, anggota tim Anda tidak terlalu gembira dengan situasi seperti ini karena tidak hanya harus merubah kode program mereka untuk menyesuaikan implementasi baru dari *class* Anda, mereka juga harus ikut mengkompilasi ulang perubahan itu.

Itulah mengapa mengimplementasikan enkapsulasi dengan membuat setiap *field* sebagai **private** dan kemudian menyediakan properti untuk mengakses *field* tersebut menjadi sangat penting.

Pada contoh kasus di atas, pada saat Anda pertama kali mendesain sebuah *class* tanpa mempertimbangkan perlu tidaknya melakukan verifikasi untuk menentukan sebuah nilai ke dalam sebuah *field*, sebaiknya jangan pernah sekalipun berpikir untuk mendeklarasikan *field* tersebut sebagai **public**. Tetap usahakan untuk mendeklarasikannya sebagai **private** dan kemudian mengeksposnya melalui sebuah properti. Sehingga, sejak awal seharusnya *class* Anda ditulis seperti berikut ini.

```
class SebuahClass
{
    private int bilBulat;
    public int BilanganBulat
    {
        get
        {
            return bilBulat;
        }

        set
        {
```

```
        bilBulat = value;  
    }  
}  
}
```

Ketika Anda mendistribusikan *class* Anda dengan desain seperti ini, pengguna dari *class* Anda mau tidak mau akan mengakses *field* `bilBulat` melalui properti `BilanganBulat`.

Dengan demikian, ketika Anda akhirnya memutuskan untuk menambahkan implementasi logika (seperti melakukan verifikasi nilai input) di dalam properti tersebut, pengguna *class* Anda yang dari awal mengakses *field* `bilBulat` melalui properti `BilanganBulat`, tidak perlu merubah setiap kode program `obyek.bilBulat` menjadi `obyek.BilanganBulat`.

Jadi, ketika Anda dituntut untuk memperbaiki implementasi *class* yang Anda buat, Anda tidak ikut memaksa pengguna *class* Anda untuk menyesuaikan perubahan yang Anda buat.

Auto-implemented property

Harus mendeklarasikan setiap *field* sebagai **private** secara eksplisit, lalu mengeksposnya melalui sebuah properti tentunya membuat kode program di dalam *class* Anda menjadi cukup panjang.

Dalam pemrograman C#, ketika Anda tidak membutuhkan implementasi logika apapun di dalam sebuah properti, Anda bisa menggunakan apa yang dinamakan dengan *auto-implemented property*. Dengan demikian, kode

program sebelumnya bisa Anda tulis seperti berikut ini.

```
class SebuahClass
{
    public int BilanganBulat { get; set; }
}
```

Perhatikan bahwa Anda bahkan sama sekali tidak perlu mendeklarasikan *field* `bilBulat`. Saat Anda mendeklarasikan properti seperti yang ditunjukkan pada kode program di atas, secara otomatis, *compiler* nantinya akan membuat sebuah *field* anonim pendukung dengan akses **private** yang hanya dapat diakses melalui aksesor `get` dan `set` pada properti tersebut.

Praktek penggunaan *auto-implemented property* seperti yang ditunjukkan pada contoh kode program di atas tentunya jauh lebih baik ketimbang mendeklarasikan sebuah *field* sebagai **public**.

Jika diperlukan, nantinya Anda tinggal mengimplementasikan logika tambahan untuk aksesor `get` dan `set` serta mendeklarasikan beberapa *field* dengan akses **private** untuk mendukung properti yang sebelumnya Anda deklarasikan. Ini bisa Anda lakukan tanpa harus memaksa pengguna *class* Anda untuk melakukan penyesuaian terhadap implementasi *class* Anda yang baru.

Mengkapsulasi beberapa fungsionalitas dengan utility class

Sejauh ini kita sudah banyak berbicara mengenai *data hiding*

(penyembunyian data) sebagai salah satu cara untuk menerapkan enkapsulasi. Namun, menerapkan enkapsulasi itu sebenarnya tidak melulu mengenai *data hiding*. Mengumpulkan beberapa fungsionalitas ke dalam sebuah *class* pun bisa dianggap sebagai cara menerapkan enkapsulasi.

Ambil contoh fungsi-fungsi matematika seperti, bilangan pangkat, pembulatan, trigonometri, logaritma, dan lain-lain. Tentunya, fungsi-fungsi tersebut tidak hanya berguna untuk satu jenis aplikasi saja. Namun, Anda bisa menggunakan fungsi-fungsi tersebut untuk berbagai keperluan di berbagai jenis aplikasi.

Agar fungsi-fungsi matematika tersebut bisa Anda gunakan kembali terus menerus untuk aplikasi-aplikasi yang Anda buat kedepannya, Anda akan lebih memilih untuk mengumpulkan fungsi-fungsi tersebut ke dalam sebuah *class library*.

Dengan mengumpulkannya ke dalam sebuah *class library*, nantinya Anda hanya perlu melakukan referensi ke *class* tersebut dan memanggil fungsinya. Tanpa harus menulis ulang kode program yang sama terus menerus untuk setiap aplikasi yang berbeda.

Class seperti ini biasa disebut dengan *utility class*. *Class* ini hanya berfungsi untuk mengumpulkan beberapa fungsionalitas dari kelompok yang sama. Contohnya, seperti mengumpulkan fungsi-fungsi matematika ke dalam sebuah *class* seperti contoh di atas.

Ketika objek menjadi ide buruk

Untuk memanfaatkan fungsi-fungsi pada sebuah *class*, Anda biasanya membuat objek dari *class* tersebut, lalu memanggil fungsi-fungsi yang terkait dengan objek tersebut.

Untuk sebuah *utility class*, ini justru menjadi masalah.

Katakan Anda mempunyai sebuah *utility class* bernama **Matematika**. *Class* ini berisi berbagai fungsi matematika.

Pada suatu saat, Anda ingin melakukan pembulatan di berbagai tempat di dalam kode program Anda. Senyum Anda merekah. Kali ini Anda hanya perlu melakukan referensi ke *class* **Matematika** dan memanggil fungsi pembulatan. Tidak perlu lagi *copy-paste* kode program yang sama.

Namun, akhirnya Anda menyadari bahwa setiap kali Anda membutuhkan fungsi pembulatan di tempat dengan *scope* yang berbeda, Anda perlu membuat objek dari *class* **Matematika**. Pada akhirnya Anda membuat belasan atau puluhan objek hanya untuk melakukan operasi pembulatan.

Anda tidak ingin mengambil ruang memori yang besar hanya untuk melakukan operasi pembulatan, bukan?

Untuk mengatasi permasalahan ini, C# memiliki solusi dengan membuat sebuah *class* menjadi *static*.

Static class

Dengan mendeklarasikan sebuah *class* sebagai **static**, artinya Anda tidak bisa membuat objek dari *class* tersebut. Seperti pada contoh sebelumnya, sangat tidak masuk akal apabila Anda harus membuat sebuah objek dari *class Matematika* terlebih dahulu agar bisa memanfaatkan fungsionalitasnya.

Yang Anda butuhkan hanyalah **fungsionalitasnya** saja, bukan karakteristik atau kondisi yang berkaitan dengan objek dari *class* tersebut.

Untuk membuat sebuah *class* menjadi *static*, Anda dapat menggunakan kata kunci **static** sebelum nama *class* seperti pada contoh berikut ini.

```
static class SebuahClass
{
}
```

Perlu diperhatikan juga apabila Anda membuat sebuah *static class* seperti yang ditunjukkan pada potongan kode program di atas, **semua** anggota penyusun *class* di dalam *class* tersebut seperti *field*, *property*, *method*, *event* dan lain-lain juga harus dideklarasikan dengan kata kunci **static**.

Dengan mendeklarasikan sebuah *class* sebagai **static**, berarti Anda mencegah siapapun bisa membuat objek dari *class* tersebut.

Alasan Anda ketika membuat *class* tersebut hanyalah untuk melakukan enkapsulasi atau mengumpulkan beberapa fungsionalitas dalam sebuah

class. Tujuannya, agar nantinya fungsi-fungsi tersebut bisa digunakan kembali tanpa harus ditulis dan dikompilasi ulang.

Untuk lebih memahami penjelasan di atas, mari kita buat sebuah *utility class* yang berisi fungsi-fungsi untuk melakukan konversi satuan suhu. Perhatikan kode program 5-6 berikut ini.

```
using System;

namespace KodeProgram_5_6
{
    static class KonversiSuhu
    {
        public static double CelsiusToFahrenheit(string suhuCelsius)
        {
            // Konversikan argument ke tipe data double
            // untuk melakukan kalkulasi.
            double celsius = Double.Parse(suhuCelsius);

            // Konversikan Celsius ke Fahrenheit.
            double fahrenheit = (celsius * 9 / 5) + 32;

            return fahrenheit;
        }

        public static double FahrenheitToCelsius(string suhuFahrenheit)
        {
            // Konversikan argument ke tipe data double
            // untuk melakukan kalkulasi.
            double fahrenheit = Double.Parse(suhuFahrenheit);

            // Konversikan Fahrenheit ke Celsius.
            double celsius = (fahrenheit - 32) * 5 / 9;
        }
    }
}
```

```

        return celsius;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Pilih jenis konversi");
        Console.WriteLine("Tulis 1 untuk konversi Dari Celsius ke Fahrenheit.");
        Console.WriteLine("Tulis 2 untuk konversi Dari Fahrenheit ke Celsius.");
        Console.Write(":");

        string pilihan = Console.ReadLine();
        double F, C = 0;

        switch (pilihan)
        {
            case "1":
                Console.Write("Masukkan suhu dalam Celsius: ");
                F = KonversiSuhu.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Suhu dalam satuan Fahrenheit: {0:F2}",
F);

                break;

            case "2":
                Console.Write("Masukkan suhu dalam Fahrenheit: ");
                C = KonversiSuhu.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Suhu dalam satuan Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Pilih 1 atau 2");
                break;
        }
    }
}

```

```

        Console.WriteLine("\nTekan Enter untuk keluar..");
        Console.ReadLine();
    }
}

```

Pada kode program 5-6 di atas, dengan membuat sebuah *class* menjadi *static*, kita dapat menggunakan fungsi `CelsiusToFahrenheit` dan `FahrenheitToCelsius` tanpa harus membuat sebuah objek dari *class* `KonversiSuhu` terlebih dahulu. Yang perlu Anda lakukan hanyalah memanggil fungsionalitasnya (*method*) melalui nama dari *class*-nya.

```
KonversiSuhu.FahrenheitToCelsius();
```

`KonversiSuhu` adalah nama *class*-nya. Sedangkan, `FahrenheitToCelsius` adalah nama *method* yang berada di dalam *class* `KonversiSuhu`.

Static members

Sebuah *class* yang dideklarasikan tanpa kata kunci `static`, akan menjadi ***non-static class***. Artinya, Anda bisa membuat sebuah objek dari *class* tersebut.

Secara default, semua *class* adalah *non-static class*. Namun demikian, Anda tetap diperkenankan untuk mendeklarasikan beberapa anggota penyusun *class* sebagai `static` meskipun mereka berada di dalam sebuah *non-static class*.

Anggota penyusun *class* yang dideklarasikan sebagai `static` di dalam

sebuah *non-static class*, **bukan merupakan karakteristik atau perilaku object** dari *class* tersebut, melainkan **karakteristik atau perilaku dari *class*** itu sendiri.

Untuk lebih memahaminya, coba perhatikan kode program berikut ini.

```
using System;

namespace KodeProgram_5_7
{
    // non-static class
    class Mobil
    {
        // anggota penyusun class yang dideklarasikan sebagai static
        // static field
        private static int JmlObjMobil = 0;

        // constructor
        public Mobil()
        {
            JmlObjMobil++;
        }

        // anggota penyusun class yang dideklarasikan sebagai static
        // static method
        public static void TotalObjMobil()
        {
            Console.WriteLine("Ada {0} mobil", JmlObjMobil);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
```



```

        Mobil.TotalObjMobil();

        Mobil sedan = new Mobil();

        Mobil.TotalObjMobil();

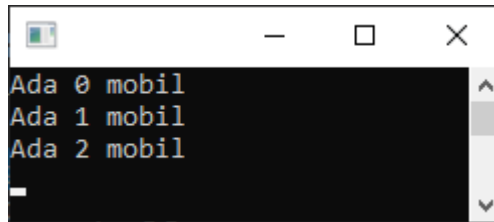
        Mobil pickup = new Mobil();

        Mobil.TotalObjMobil();

        Console.ReadLine();
    }
}

```

Kode program di atas akan menghasilkan keluaran seperti pada Gambar 5.1.



Gambar 5.1

Pada Kode Program 5-7, *class* `Mobil` adalah sebuah *non-static class*. Namun demikian, beberapa anggota penyusun *class* seperti *field* `JmlObjMobil` dan *method* `TotalObjMobil()`, kita deklarasikan sebagai **static**.

Artinya, meskipun kita bisa membuat beberapa objek dari *class* tersebut seperti objek `sedan` dan objek `pickup`, anggota penyusun *class* yang dideklarasikan sebagai **static** bukanlah merupakan karakteristik dan

perilaku dari objek-objek yang kita buat tadi, melainkan merupakan karakteristik dan perilaku dari *class* itu sendiri.

Buktinya, sebelum kita membuat objek *sedan*, *field* `JmlObjMobil` bernilai 0. Lalu, pada saat objek *sedan* dibuat, nilai `JmlObjMobil` dinaikkan satu di dalam *constructor class* `Mobil`. Dengan demikian, setelah objek *sedan* dibuat, nilai `JmlObjMobil` menjadi bernilai 1.

Demikian juga ketika kita sekali lagi membuat sebuah objek dari *class* `Mobil` bernama *pickup*. Di dalam *constructor class* `Mobil`, nilai `JmlObjMobil` akan dinaikkan satu nilai lagi sehingga kini bernilai 2.

Hal ini menjelaskan bahwa dengan membuat *field* `JmlObjMobil` sebagai *static*, maka di dalam memori, hanya terdapat satu saja salinan dari *field* tersebut, tidak peduli berapapun jumlah objek yang dibuat dari *class* `Mobil`. Hal itu dikarenakan *field* tersebut hanya berkaitan dengan *class* `Mobil` dan bukan pada objek *sedan* maupun *pickup*.

Perlu diperhatikan juga bahwa dengan mendeklarasikan sebuah *method* sebagai *static*, maka *method* tersebut bisa diakses melalui nama *class* di mana *method* tersebut dideklarasikan. Sehingga, *method* `TotalObjMobil()` dapat kita panggil dengan cara seperti ini

```
Mobil.TotalObjMobil();
```

Perlu diperhatikan juga bahwa jika sebuah *method* dideklarasikan sebagai *static*, maka *method* tersebut hanya bisa menerima *field* yang juga

dideklarasikan sebagai **static**.

Misalnya, apabila *field* `JmlObjMobil` tidak kita deklarasikan sebagai **static**, maka ketika *field* tersebut digunakan di dalam *method* `TotalJmlMobil()` yang dideklarasikan sebagai **static**, *compiler* akan memberi Anda pesan error.

```
// non-static member
private int JmlObjMobil;

public static void TotalJmlMobil()
{
    // baris kode program ini menghasilkan error
    // JmlObjMobil harus dideklarasikan sebagai static
    Console.WriteLine(JmlObjMobil);
}
```

Sekarang, coba perhatikan kode program berikut ini, lalu bandingkan keluarannya dengan keluaran yang diperoleh dari Kode Program 5-7.

```
using System;

namespace KodeProgram_5_8
{
    class Mobil
    {
        private int JmlObjMobil = 0;

        public Mobil()
        {
            JmlObjMobil++;
        }

        public void TotalObjMobil()
        {
            Console.WriteLine("Ada {0} mobil", JmlObjMobil);
        }
    }
}
```

```

    }

    class Program
    {
        static void Main(string[] args)
        {
            // Method non-static tidak bisa dipanggil dengan cara ini
            // Mobil.TotalObjMobil();

            Mobil sedan = new Mobil();

            // Mobil.TotalObjMobil();
            // method TotalObjMobil() yang terkait dengan objek sedan
            sedan.TotalObjMobil();

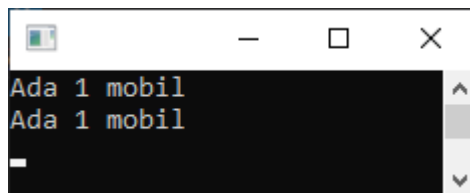
            Mobil pickup = new Mobil();

            // Mobil.TotalObjMobil();
            // method TotalObjMobil() yang terkait dengan objek pickup
            pickup.TotalObjMobil();

            Console.ReadLine();
        }
    }
}

```

Keluaran dari kode program 5-8 di atas adalah seperti berikut ini.



Gambar 5.2

Mengapa keluarannya berbeda dengan kode program sebelumnya?

Karena kini *field* `JmlObjMobil` merupakan *non-static field*, maka *field* tersebut akan menjadi terkait dengan masing-masing objek yang dibuat dari *class* `Mobil`.

Sederhananya, ketika kita membuat objek `sedan`, maka satu salinan dari *field* `JmlObjMobil` akan dialokasikan ke sebuah lokasi di memori untuk objek `sedan`. Demikian juga ketika kita membuat objek `pickup`, maka satu lagi salinan dari *field* `JmlObjMobil` akan dialokasikan untuk objek `pickup` di dalam memori dengan lokasi yang berbeda.

Dengan demikian, salinan *field* `JmlObjMobil` untuk objek `sedan` dan salinan *field* `JmlObjMobil` untuk objek `pickup` adalah dua salinan yang berbeda karena terletak di dua lokasi memori yang berbeda.

Sehingga perlu diingat bahwa setiap kali kita membuat sebuah objek dari *class* `Mobil`, maka satu salinan dari *field* `JmlObjMobil` yang berkaitan dengan objek tersebut juga akan dialokasikan ke sebuah lokasi di dalam memori.

Jadi, katakanlah kita membuat lima objek dari *class* `Mobil`, maka lima salinan *field* `JmlObjMobil` untuk masing-masing objek tersebut juga akan dialokasikan ke dalam memori.

Oleh karena itu, meskipun nilai *field* `JmlObjMobil` dinaikkan satu di dalam *constructor* setiap kali sebuah objek dibuat, karena salinan *field*

`JmlObjMobil` untuk setiap objek berada pada lokasi memori yang berbeda, maka nilai *field* tersebut selamanya hanya akan bernilai 1 untuk masing-masing objek.

Static constructor

Apabila di bab 4 Anda sudah mempelajari apa itu *default constructor*, *parameterized constructor*, dan *copy constructor*, sekarang kita akan mempelajari apa itu *static constructor*.

Sama halnya dengan *static member*, sebuah *static constructor* juga bisa dideklarasikan di dalam sebuah *non-static class*.

Static constructor digunakan untuk menginisialisasi *static member* pada sebuah *class* atau untuk menjalankan sebuah operasi yang hanya perlu dijalankan sekali.

Static constructor akan dipanggil secara otomatis pada dua kondisi. Yang pertama adalah sebelum objek pertama dari *class*-nya dibuat. Sedangkan yang kedua adalah sebelum sebuah *static member* dari *class*-nya digunakan (referensinya).

Kode program 5-9 di bawah ini mendemonstrasikan bagaimana *static constructor* dipanggil secara otomatis pada kondisi pertama.

```
using System;

namespace KodeProgram_5_9
{
```

```

class Mobil
{
    public string pabrik;
    public string tipe;
    public int thnPembuatan;
    public static int kecepatanMax;

    // static constructor
    static Mobil()
    {
        kecepatanMax = 200; //km per jam
        Console.WriteLine("Kecepatan maximum pada speedometer adalah {0}",
kecepatanMax);
    }

    // Parameterized Constructor
    public Mobil(string pabrik, string model, int thnPembuatan)
    {
        this.pabrik = pabrik;
        this.model = model;
        this.thnPembuatan = thnPembuatan;
    }
}

class Program
{
    static void Main(string[] args)
    {
        // objek mobilToyota
        Console.WriteLine("--- objek mobilToyota1 ---");
        Mobil mobilToyota = new Mobil("Toyota", "Hatchback", 2019);
        Console.WriteLine(mobilToyota.pabrik);
        Console.WriteLine(mobilToyota.tipe);
        Console.WriteLine(mobilToyota.thnPembuatan);

        // objek mobilMazda
        Console.WriteLine("--- objek mobilToyota1 ---");
        Mobil mobilMazda = new Mobil("Mazda", "SUV", 2019);
    }
}

```

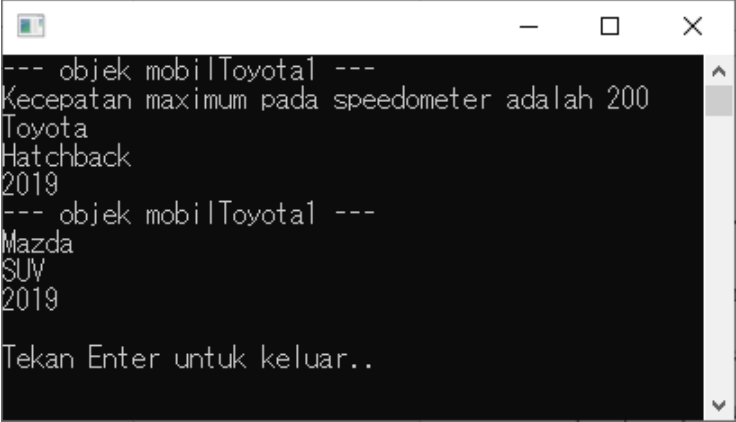
```

        Console.WriteLine(mobilMazda.pabrikan);
        Console.WriteLine(mobilMazda.tipe);
        Console.WriteLine(mobilMazda.thnPembuatan);

        Console.WriteLine("\nTekan Enter untuk keluar..");
        Console.ReadLine();
    }
}

```

Kode program 5-9 di atas akan menghasilkan keluaran seperti yang ditampilkan pada Gambar 5.3 di bawah ini.



```

--- objek mobilToyota ---
Kecepatan maximum pada speedometer adalah 200
Toyota
Hatchback
2019
--- objek mobilToyota ---
Mazda
SUV
2019
Tekan Enter untuk keluar..

```

Gambar 5.3

Jika Anda perhatikan kembali kode program 5-9 di atas, *constructor* yang dipanggil saat instansiasi objek mobilToyota sebenarnya adalah `public Mobil(string pabrikan, string model, int thnPembuatan)`. Namun, karena *class* `Mobil` memiliki *static constructor*, maka CLR akan memanggil *static constructor*-nya terlebih dahulu sebelum objek pertama dari *class* `Mobil` dibuat.

Pada saat objek kedua (mobilMazda) diinstansiasi, *static constructor* tidak lagi dipanggil. Maka dari itu kita hanya bisa melihat tulisan “Kecepatan maximum pada speedometer adalah 200” sekali saja.

Selanjutnya kode program 5-10 di bawah ini mendemonstrasikan bagaimana *static constructor* dipanggil pada kondisi kedua.

```
using System;
```

```
namespace KodeProgram_5_10
```

```
{
```

```
    class Mobil
```

```
    {
```

```
        public string pabrikan;
```

```
        public string tipe;
```

```
        public int thnPembuatan;
```

```
        public static int kecepatanMax;
```

```
        // static constructor
```

```
        static Mobil()
```

```
        {
```

```
            kecepatanMax = 200; //km per jam
```

```
            Console.WriteLine("Kecepatan maximum pada speedometer adalah {0}",  
kecepatanMax);
```

```
        }
```

```
    }
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int max = Mobil.kecepatanMax;
```

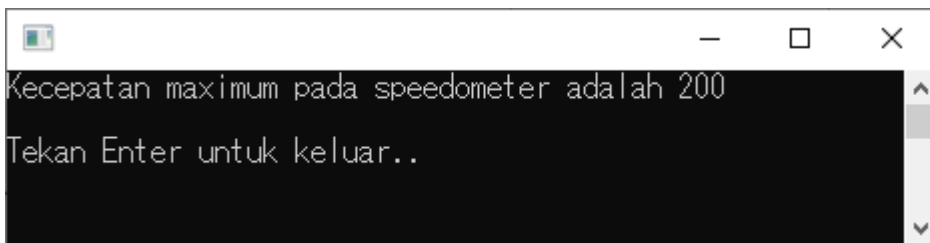
```
            Console.WriteLine("Tekan Enter untuk keluar..");
```

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

```
    }  
    }  
}
```

Kali ini di dalam *method* `Main()`, kita tidak membuat satu pun objek dari *class* `Mobil`. Kita hanya menggunakan salah satu *static member* dari *class* `Mobil` yaitu `kecepatanMax`. Instruksi ini juga akan membuat *static constructor* dipanggil secara otomatis oleh CLR.

Kode program 5-10 di atas akan menghasilkan keluaran seperti yang ditampilkan pada Gambar 5.4.



Gambar 5.4

Ada beberapa hal yang perlu Anda perhatikan pada saat mendeklarasikan dan menggunakan *static constructor*, yaitu

- 1) Anda tidak bisa memberi *access modifier* (*public*, *private*, dll) apapun kepada *static constructor*.
- 2) *Static constructor* tidak bisa memiliki parameter.
- 3) Anda tidak mempunyai kontrol terhadap kapan *static constructor* dipanggil. Pemanggilan *static constructor* sepenuhnya berada dalam

kontrol CLR.

4) Anda tidak bisa memanggilnya secara langsung.

Private constructor

Sama halnya ketika Anda mendeklarasikan sebuah *class* dengan kata kunci **static**, mendeklarasikan *constructor* di dalam sebuah *class* sebagai **private** juga akan mencegah pembuatan objek atau instansiasi dari *class* tersebut.

```
public class KonversiSuhu
{
    private KonversiSuhu()
    {

    }
}

class Program
{
    static void Main(string[] args)
    {
        // Instansiasi objek dari class KonversiSuhu
        // akan menghasilkan eror
        KonversiSuhu utilityObj = new KonversiSuhu();
    }
}
```

Lalu apa bedanya dengan *static class*?

Pada dasarnya baik membuat *constructor* sebuah *class* sebagai **private** maupun membuat sebuah *class* menjadi *static* merupakan hal yang sama

dengan tujuan yang sama.

Lalu kapan sebaiknya menggunakan *private constructor*?

Saya pribadi tidak pernah menggunakan *private constructor* ketika menulis kode program dan lebih memilih menggunakan *static class*.

Sebenarnya *private constructor* dulunya digunakan oleh para *developer .NET* ketika *static class* belum diperkenalkan di platform *.NET* kala itu. Sehingga untuk mencegah pembuatan objek dari sebuah *class*, mereka membuat *constructor class* tersebut sebagai **private**.

Bab 6

Mewariskan Karakteristik Dan Perilaku Sebuah Class

Keluarga objek

Banyak hal di dunia ini yang tergabung dalam sebuah grup yang berkaitan ataupun sebuah keluarga. Sesuatu yang tergabung dalam sebuah grup atau keluarga yang sama biasanya memiliki karakteristik atau perilaku yang hampir mirip.

Begitupun di dalam pemrograman berorientasi objek. Anda nantinya akan menyadari bahwa beberapa objek memiliki karakteristik dan perilaku yang hampir sama. Akan lebih baik jika kita dapat mengelompokkan objek-objek tersebut ke dalam sebuah keluarga objek yang sama.

Generalisasi dan spesialisasi

Masih ingat pelajaran biologi tentang klasifikasi ilmiah yang mengelompokkan makhluk hidup dari tingkat kerajaan sampai spesies?

Dalam biologi, makhluk hidup dikelompokkan dari karakteristik dan perilaku yang paling umum sampai karakteristik dan perilaku yang paling spesifik. Inilah yang disebut dengan generalisasi dan spesialisasi.

Coba Anda perhatikan pengelompokan makhluk hidup berikut ini sambil mengingat kembali pelajaran biologi yang pernah Anda pelajari di bangku sekolah.

Kerajaan	Binatang
Filum	Vertebrata
Kelas	Mamalia
Ordo	Karnivora
Famili	Felidae
Genus	Felis
Spesies	Felis Catus (kucing)

Katakan Anda memiliki seekor kucing yang Anda beri nama “Nyanyan”, maka “Nyanyan” ini bukan termasuk nama klasifikasi makhluk hidup, melainkan nama dari seekor binatang dari spesies kucing. Dalam dunia pemrograman berorientasi objek, “Nyanyan” adalah sebuah objek dari *class* Felis Catus atau kucing.

Pada contoh pengelompokan makhluk hidup di atas, tingkat klasifikasi di atasnya merupakan generalisasi dari tingkat di bawahnya. Sedangkan tingkat klasifikasi dibawahnya merupakan spesialisasi dari tingkat di atasnya.

Contohnya, karnivora adalah spesialisasi dari mamalia. Sehingga setiap

karnivora memiliki karakteristiknya sendiri dan juga sekaligus mewarisi karakteristik dari mamalia, namun setiap mamalia belum tentu memiliki setiap karakteristik yang dimiliki oleh karnivora.

Pewarisan dalam pemrograman berorientasi objek

Membuat sebuah *class* memiliki seluruh karakteristik dan perilaku dari *class* yang lebih umum (*general*) disebut dengan pewarisan.

Mari kita perhatikan apa saja karakteristik dan perilaku yang dimiliki oleh sebuah mobil dan sebuah motor.

Mobil	Motor
<i>Karakteristik</i>	<i>Karakteristik</i>
Jumlah Roda Bahan Bakar Nomor Polisi Transmisi	Jumlah Roda Bahan Bakar Nomor Polisi Transmisi
<i>Perilaku</i>	<i>Perilaku</i>
Hidupkan Mesin Matikan Mesin Jalan	Hidupkan Mesin Matikan Mesin Jalan

Gambar 6.1

Tentu saja daftar di atas tidak mencakup semua karakteristik dan perilaku yang dimiliki oleh sebuah mobil dan motor. Saya hanya menuliskan beberapa saja agar daftarnya menjadi lebih sederhana.

Sekarang, coba bandingkan daftar karakteristik dan juga perilaku yang dimiliki oleh sebuah mobil dan sebuah motor. Apabila kita perhatikan, ada beberapa karakteristik dan perilaku yang sama-sama dimiliki baik oleh sebuah mobil maupun oleh sebuah motor.

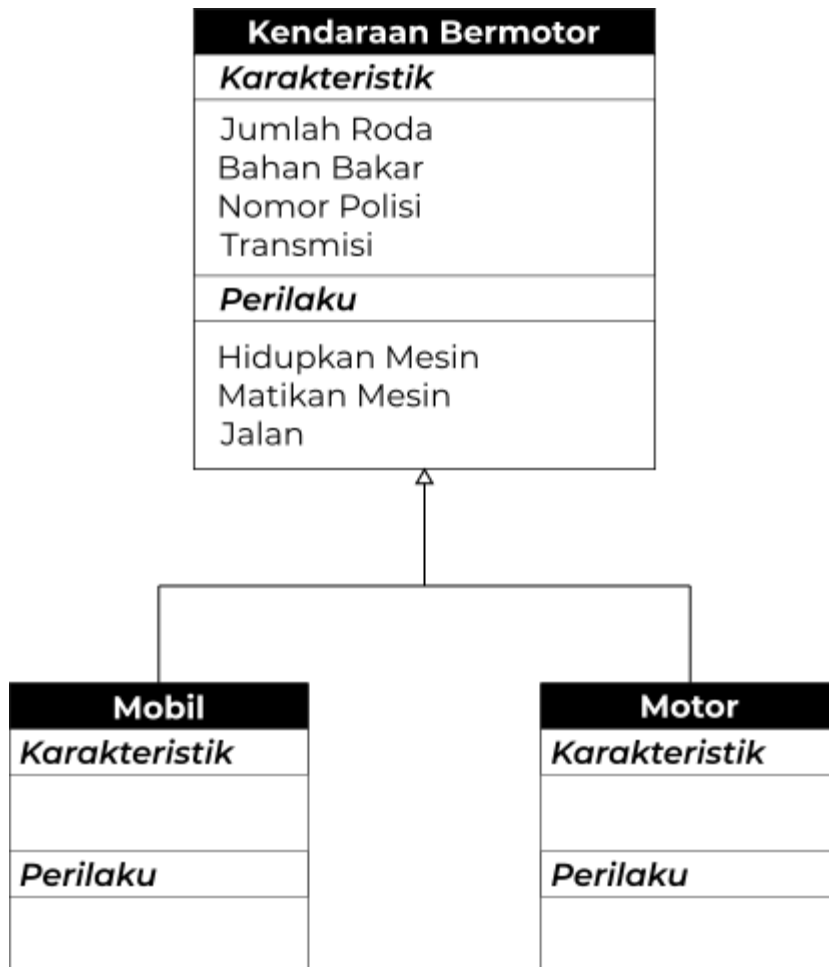
Sehingga secara umum, Anda bisa mengklasifikasikan karakteristik dan perilaku yang sama tadi sebagai karakteristik dan perilaku dari sebuah bentuk yang lebih umum daripada mobil atau motor, yaitu kendaraan bermotor.

Kendaraan Bermotor
<i>Karakteristik</i>
Jumlah Roda Bahan Bakar Nomor Polisi Transmisi
<i>Perilaku</i>
Hidupkan Mesin Matikan Mesin Jalan

Gambar 6.2

Setelah Anda melakukan generalisasi bentuk dari sebuah mobil dan motor, Anda dapat melakukan spesialisasi bentuk untuk sebuah mobil dan motor. Maksudnya, selanjutnya Anda hanya perlu membuat daftar karakteristik dan perilaku yang spesifik dimiliki oleh sebuah mobil dan juga membuat daftar karakteristik dan perilaku yang spesifik dimiliki oleh sebuah motor.

Dengan demikian, Anda bisa menggambarkan hirarki mobil dan motor dengan kendaraan bermotor seperti berikut ini.



Gambar 6.3

Diagram di atas menyatakan bahwa baik mobil maupun motor dapat memiliki karakteristik dan perilakunya masing-masing, sekaligus mewarisi semua karakteristik dan perilaku yang dimiliki oleh kendaraan bermotor.

Mengimplementasikan konsep pewarisan di C#

Seperti pada contoh sebelumnya, kendaraan bermotor merupakan klasifikasi yang lebih umum dari bentuk yang lebih spesifik seperti mobil dan motor. Dalam pemrograman C#, kendaraan bermotor ini biasa disebut dengan *superclass* atau *base class* (*class* dasar) sedangkan mobil dan motor biasa disebut dengan *subclass* atau *derived class* (*class* turunan).

Untuk mengimplementasikan pewarisan di C#, Anda hanya perlu menambahkan simbol titik dua (:) setelah nama *derived class* lalu diikuti dengan nama *base class*-nya seperti berikut ini.

```
class NamaDerivedClass : NamaBaseClass
{
}
```

Sehingga, apabila Anda ingin mengimplementasikan diagram pada Gambar 6.3 ke dalam beberapa *class*, maka Anda bisa menuliskannya seperti kode program berikut ini.

```
using System;

namespace KodeProgram_6_1
{
    class KendaraanBermotor
    {
        public int JmlRoda { get; set; }
        public string BahanBakar { get; set; }
        public string NoPolisi { get; set; }
    }
}
```

```

public string Transmisi { get; set; }

public void HidupkanMesin()
{
    Console.WriteLine("Mesin Kendaraan dihidupkan");
}

public void MatikanMesin()
{
    Console.WriteLine("Mesin Kendaraan dimatikan");
}

public void Jalan()
{
    Console.WriteLine("Berjalan");
}
}

class Mobil : KendaraanBermotor
{
    public Mobil()
    {
        this.JmlRoda = 4;
        this.BahanBakar = "Pertalite";
        this.Transmisi = "Otomatis";
    }
}

class Motor : KendaraanBermotor
{
    public Motor()
    {
        this.JmlRoda = 2;
        this.BahanBakar = "Premium";
        this.Transmisi = "Otomatis";
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        // instantiasi objek mobil
        Mobil mobil = new Mobil();

        mobil.NoPolisi = "AB 1234 CD";

        Console.WriteLine("Objek Mobil:");
        Console.WriteLine("Jumlah roda: {0}", mobil.JmlRoda);
        Console.WriteLine("Bahan bakar: {0}", mobil.BahanBakar);
        Console.WriteLine("Nomor polisi: {0}", mobil.NoPolisi);
        Console.WriteLine("Transmisi: {0}", mobil.Transmisi);

        mobil.HidupkanMesin();
        mobil.Jalan();
        mobil.MatikanMesin();

        // instantiasi objek motor
        Motor motor = new Motor();

        motor.NoPolisi = "AB 5678 CD";

        Console.WriteLine("\nObjek Motor:");
        Console.WriteLine("Jumlah roda: {0}", motor.JmlRoda);
        Console.WriteLine("Bahan bakar: {0}", motor.BahanBakar);
        Console.WriteLine("Nomor Polisi: {0}", motor.NoPolisi);
        Console.WriteLine("Transmisi: {0}", motor.Transmisi);

        motor.HidupkanMesin();
        motor.Jalan();
        motor.MatikanMesin();

        Console.ReadLine();
    }
}

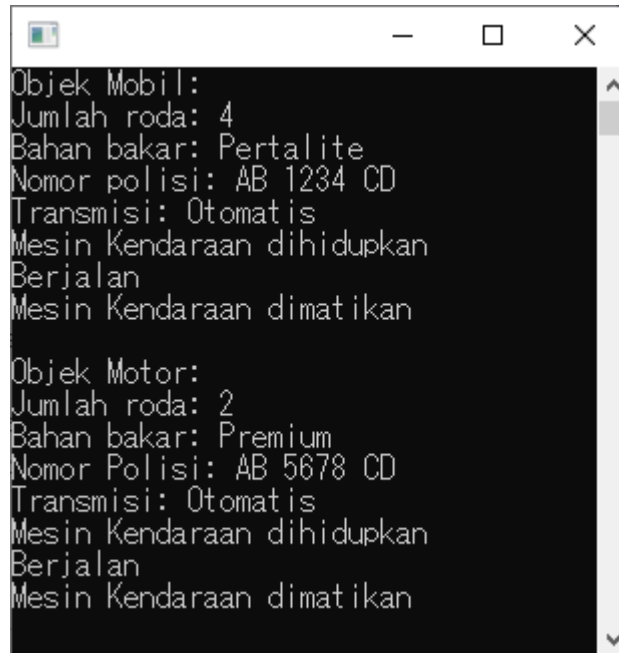
```

```

    }
}

```

Kode program di atas akan menghasilkan keluaran seperti pada Gambar 6.4.



```

Objek Mobil:
Jumlah roda: 4
Bahan bakar: Pertalite
Nomor polisi: AB 1234 CD
Transmisi: Otomatis
Mesin Kendaraan dihidupkan
Berjalan
Mesin Kendaraan dimatikan

Objek Motor:
Jumlah roda: 2
Bahan bakar: Premium
Nomor Polisi: AB 5678 CD
Transmisi: Otomatis
Mesin Kendaraan dihidupkan
Berjalan
Mesin Kendaraan dimatikan

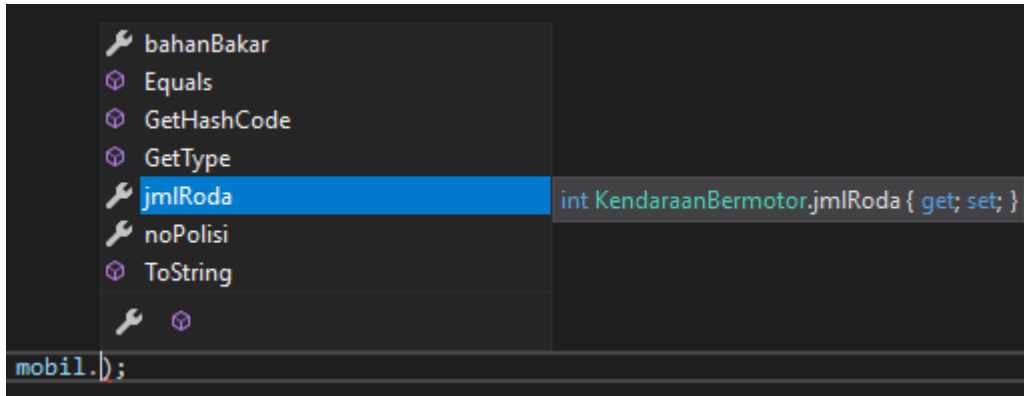
```

Gambar 6.4

Perhatikan bahwa dengan membuat objek `mobil` dari class `Mobil` dan objek `motor` dari class `Motor`, kita bahkan bisa mengakses seluruh properti dan *method* dari class `KendaraanBermotor` melalui kedua objek tersebut. Ini dimungkinkan karena class `Mobil` yang menjadi *blueprint* dari objek `mobil` dan class `Motor` yang menjadi *blueprint* dari objek `motor` mewarisi seluruh karakteristik dan perilaku dari class `KendaraanBermotor`.

Di Visual Studio, apabila Anda menuliskan nama objeknya, yaitu `mobil`,

diikuti oleh tanda titik (.), maka fitur *intellisense* akan menampilkan semua anggota penyusun *class* `KendaraanBermotor` yang dapat diakses oleh objek `mobil` seperti yang bisa Anda lihat pada Gambar 6.5.



Gambar 6.5

Salah satu keuntungan dari melakukan generalisasi dan spesialisasi, lalu mengimplementasikan pewarisan seperti di atas adalah, Anda dapat menghindari duplikasi kode yang Anda gunakan baik untuk *class* `Mobil` maupun *class* `Motor`.

Pewarisan pada constructor

Dalam pemrograman C#, baik *base class* maupun *derived class*-nya dapat memiliki *constructor*-nya masing-masing. *Constructor* pada *base class* digunakan untuk menginstansiasi (membuat) objek dari *base class*, sedangkan *constructor* pada *derived class* digunakan untuk menginstansiasi objek dari *derived class*.

Dalam pewarisan, sebuah *derived class* akan mewarisi semua anggota penyusun *class* (*field*, properti, *method*) pada *base class*-nya, tetapi **sebuah *derived class* tidak dapat mewarisi *constructor* pada *base class*-nya** karena *constructor* bukanlah merupakan anggota penyusun *class*. Meski demikian, sebuah *derived class* dimungkinkan untuk memanggil *constructor* pada *base class*-nya.

Dalam pemrograman C#, ada beberapa skenario yang mungkin muncul ketika Anda bekerja dengan *constructor* dalam pewarisan, antara lain:

Skenario 1: Anda mendeklarasikan *constructor* tanpa parameter pada *base class* tanpa mendeklarasikan *constructor* pada *derived class*.

Pada skenario seperti ini, ketika Anda membuat objek dari sebuah *derived class*, *constructor* pada *base class*-nya akan dipanggil secara otomatis.

```
using System;
```

```
namespace KodeProgram_6_2
{
    class KendaraanBermotor
    {
        public int JmlRoda { get; set; }
        public string BahanBakar { get; set; }
        public string NoPolisi { get; set; }
        public string Transmisi { get; set; }

        // constructor tanpa parameter pada base class
        public KendaraanBermotor ()
```

```

        {
            this.JmlRoda = 4;
            this.BahanBakar = "Pertalite";
            this.NoPolisi = "AB 1234 CD";
            this.Transmisi = "Otomatis";
        }
    }

    class Mobil : KendaraanBermotor
    {
        // tidak ada deklarasi constructor secara eksplisit di sini
    }

    class Program
    {
        static void Main(string[] args)
        {
            Mobil mobil = new Mobil();

            Console.WriteLine("Jumlah roda: {0}", mobil.JmlRoda);
            Console.WriteLine("Bahan bakar: {0}", mobil.BahanBakar);
            Console.WriteLine("Nomor Polisi: {0}", mobil.NoPolisi);
            Console.WriteLine("Transmisi: {0}", mobil.Transmisi);

            Console.ReadLine();
        }
    }
}

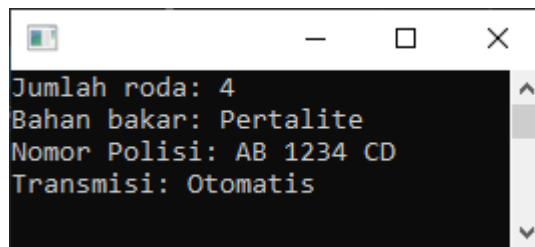
```

Perlu diingat bahwa meskipun Anda tidak mendeklarasikan sebuah *constructor* secara eksplisit di dalam sebuah *class*, *compiler* secara otomatis tetap akan menyediakan sebuah *constructor* tanpa parameter atau *default constructor* dengan akses **public** untuk *class* tersebut.

Constructor `Mobil()` adalah *default constructor* pada *class* `Mobil` yang secara otomatis akan disediakan oleh *compiler* apabila Anda tidak mendeklarasikannya secara eksplisit di dalam *class* `Mobil`.

Perhatikan kembali kode program 6-2 di atas. Ketika objek `mobil` dibuat melalui *default constructor* pada *class* `Mobil` (`new Mobil()`), *constructor* tersebut secara otomatis akan memanggil *constructor* pada *class* `KendaraanBermotor`. Selanjutnya, nilai properti `JmlRoda`, `BahanBakar`, `NoPolisi`, dan `Transmisi` akan diinisialisasi di dalam *constructor* pada *class* `KendaraanBermotor`.

Kode program di atas menghasilkan keluaran seperti pada Gambar 6.6.



Gambar 6.6

Meskipun kode program pada skenario ini dapat dijalankan tanpa adanya eror, sebaiknya desain seperti ini Anda hindari karena menginisialisasi data-data objek dari sebuah *derived class* di dalam *constructor* pada *base class*-nya merupakan praktek pemrograman yang kurang baik.

Skenario 2: Anda mendeklarasikan *constructor* pada *derived class* tanpa mendeklarasikan *constructor* pada *base class*.

Pada skenario ini, apabila Anda membuat objek dari sebuah *derived class*, maka objek tersebut akan diinstansiasi oleh *constructor* pada *derived class* tersebut. Perhatikan bahwa *field* atau properti yang dideklarasikan di *base class*-nya pun bisa Anda inisialisasi di dalam *constructor* pada *derived class*.

Sedangkan, apabila Anda membuat sebuah objek dari *base class*, maka objek tersebut akan diinstansiasi secara otomatis oleh *default constructor* pada *base class*. Namun demikian, *field-field* yang dideklarasikan di *base class* tidak akan diinisialisasi.

`using System;`

```
namespace KodeProgram_6_3
{
    class KendaraanBermotor
    {
        public int JmlRoda { get; set; }
        public string BahanBakar { get; set; }
        public string NoPolisi { get; set; }
        public string Transmisi { get; set; }
    }

    class Mobil : KendaraanBermotor
    {
        // constructor tanpa parameter pada derived class
        public Mobil()
        {
            this.JmlRoda = 4;
            this.BahanBakar = "Pertalite";
            this.NoPolisi = "AB 1234 CD";
            this.Transmisi = "Otomatis";
        }
    }
}
```

```

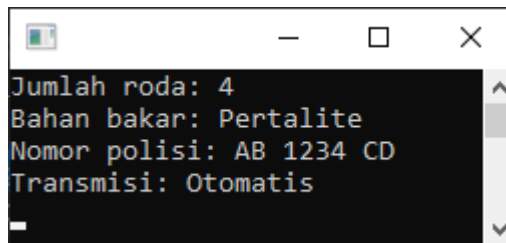
class Program
{
    static void Main(string[] args)
    {
        Mobil mobil = new Mobil();

        Console.WriteLine("Jumlah roda: {0}", mobil.JmlRoda);
        Console.WriteLine("Bahan bakar: {0}", mobil.BahanBakar);
        Console.WriteLine("Nomor polisi: {0}", mobil.NoPolisi);
        Console.WriteLine("Transmisi: {0}", mobil.Transmisi);

        Console.ReadLine();
    }
}

```

Program di atas menghasilkan keluaran seperti pada Gambar 6.7.



Gambar 6.7

Skenario 3: Anda mendeklarasikan *constructor* tanpa parameter secara eksplisit baik untuk *base class* maupun *derived class*.

Pada skenario ini, apabila Anda membuat objek dari sebuah *derived class*, maka *constructor* pada *derived class* secara otomatis akan memanggil *constructor* pada *base class*-nya terlebih dahulu sebelum menjalankan instruksi-instruksi di dalam *constructor*-nya sendiri.

```
using System;

namespace KodeProgram_6_4
{
    class KendaraanBermotor
    {
        public int JmlRoda { get; set; }
        public string BahanBakar { get; set; }
        public string NoPolisi { get; set; }
        public string Transmisi { get; set; }

        // constructor tanpa parameter pada base class
        public KendaraanBermotor()
        {
            Console.WriteLine("Sebuah kendaraan bermotor berhasil dibuat");
            Console.WriteLine("Detail kendaraan:");
        }
    }

    class Mobil : KendaraanBermotor
    {
        // constructor tanpa parameter pada derived class
        public Mobil()
        {
            this.JmlRoda = 4;
            this.BahanBakar = "Pertalite";
            this.NoPolisi = "AB 1234 CD";
            this.Transmisi = "Otomatis";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Mobil mobil = new Mobil();
        }
    }
}
```

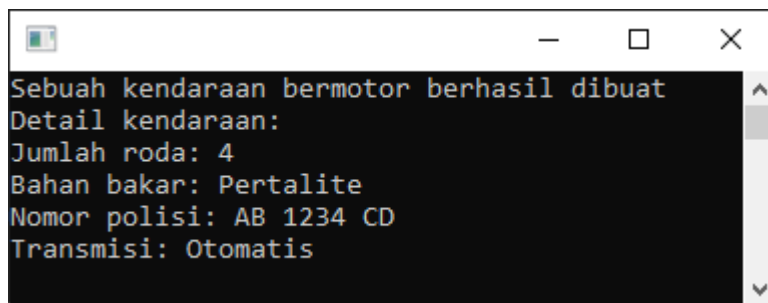
```

        Console.WriteLine("Jumlah roda: {0}", mobil.JmlRoda);
        Console.WriteLine("Bahan bakar: {0}", mobil.BahanBakar);
        Console.WriteLine("Nomor polisi: {0}", mobil.NoPolisi);
        Console.WriteLine("Transmisi: {0}", mobil.Transmisi);

        Console.ReadLine();
    }
}

```

Program di atas menghasilkan keluaran seperti pada Gambar 6.8.



Gambar 6.8

Perhatikan bahwa tulisan "Sebuah kendaraan bermotor berhasil dibuat" dan "Detail kendaraan:", yang merupakan keluaran dari instruksi di dalam *constructor* pada class `KendaraanBermotor`, ditampilkan sebelum data-data dari objek `mobil`.

Skenario 4: Anda mendeklarasikan *constructor* dengan parameter pada *base class*.

Pada skenario ini, ketika Anda membuat sebuah objek dari sebuah *derived class*, *constructor* pada *derived class* secara otomatis tidak bisa memanggil

constructor pada *base class*.

Untuk mengatasi masalah ini, C# menyediakan kata kunci **base**. Dengan bantuan kata kunci **base**, *constructor* pada *derived class* bisa memanggil *constructor* dengan parameter pada *base class*-nya.

Berikut ini adalah sintaks untuk menambahkan kata kunci **base** pada *constructor derived class*:

```
derived-constructor(daftar-parameter) : base(daftar-argument)
{
    // instruksi di dalam constructor
}
```

Daftar argument pada **base** harus sesuai dengan daftar argument di dalam *constructor* yang berada di *base class*-nya seperti pada kode program berikut ini.

```
using System;

namespace KodeProgram_6_5
{
    class KendaraanBermotor
    {
        public int JmlRoda { get; set; }
        public string BahanBakar { get; set; }
        public string NoPolisi { get; set; }
        public string Transmisi { get; set; }

        // constructor dengan satu buah parameter pada base class
        public KendaraanBermotor(string transmisi)
        {
            this.Transmisi = transmisi;
        }
    }
}
```

```

        Console.WriteLine("Transmisi: {0}", this.Transmisi);
    }
}

class Mobil : KendaraanBermotor
{
    // constructor dengan parameter pada derived class
    // argumen yang diberikan pada base harus sesuai jumlah dan tipenya
    // dengan parameter pada constructor di base class
    public Mobil(string transmisi) : base(transmisi)
    {
        this.JmlRoda = 4;
        this.BahanBakar = "Pertalite";
        this.NoPolisi = "AB 1234 CD";
    }
}

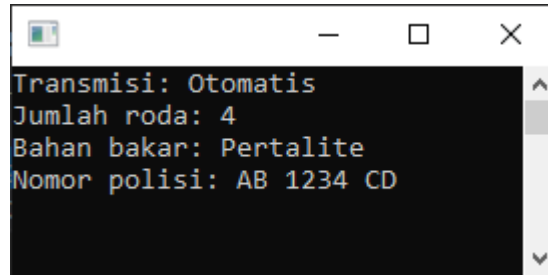
class Program
{
    static void Main(string[] args)
    {
        Mobil mobil = new Mobil("Otomatis");

        Console.WriteLine("Jumlah roda: {0}", mobil.JmlRoda);
        Console.WriteLine("Bahan bakar: {0}", mobil.BahanBakar);
        Console.WriteLine("Nomor polisi: {0}", mobil.NoPolisi);

        Console.ReadLine();
    }
}

```

Program di atas menghasilkan keluaran seperti pada Gambar 6.9.



Gambar 6.9

Anggota class dengan akses **private** tidak bisa diwariskan

Seperti yang sudah Anda ketahui, sebuah anggota penyusun *class* dengan akses **private** hanya bisa diakses oleh kode program yang berada pada *class* yang sama. Artinya, *class* turunannya sekalipun tidak diizinkan untuk mengakses anggota penyusun *class* tersebut.

Agar lebih mudah memahaminya, mari kita modifikasi kode program 6-5 menjadi seperti berikut ini.

```
using System;

namespace KodeProgram_6_6
{
    class KendaraanBermotor
    {
        public string Transmisi;

        public KendaraanBermotor(string transmisi)
        {
            this.Transmisi = transmisi;
        }
    }
}
```



```

    public void PindahTransmisi()
    {
        Console.WriteLine("Rubah rasio putaran turbin");
    }
}

class Mobil : KendaraanBermotor
{
    public Mobil(string transmisi): base (transmisi)
    {
    }

    public void JalanMaju()
    {
        if (Transmisi == "Otomatis")
        {
            Console.WriteLine("Pindah tuas transmisi dari N ke D");
            Console.WriteLine("Injak pedal gas");
            PindahTransmisi();
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        Mobil mobil = new Mobil("Otomatis");

        // sebuah mobil tidak bisa menentukan apa transmisinya
        // setelah dibuat
        mobil.Transmisi = "Otomatis";

        mobil.JalanMaju();
    }
}

```

```
// sebuah mobil tidak bisa tiba-tiba memindah transmisinya
mobil.PindahTransmisi();

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

Kali ini kita mendeklarasikan sebuah *field* bernama *Transmisi* dengan akses **public** dan juga sebuah *method* bernama *PindahTransmisi()* yang juga memiliki akses **public**.

Alasan kita memberi akses **public** untuk dua anggota penyusun *class* tersebut adalah karena kita ingin kedua anggota penyusun *class* tersebut bisa diwariskan ke *class* turunannya. Dengan begitu, *class Mobil* yang mewarisi anggota *class KendaraanBermotor* bisa menggunakan *field Transmisi* dan *method PindahTransmisi*.

Kita tidak bisa melakukan ini apabila kedua anggota penyusun *class* tersebut dideklarasikan dengan akses **private** karena meskipun *class Mobil* merupakan *class* turunan dari *class KendaraanBermotor*, namun *class Mobil* dan *class KendaraanBermotor* sebenarnya adalah dua *class* yang berbeda.

Masalah sebenarnya adalah dengan memberi akses **public** untuk *field Transmisi* dan *method PindahTransmisi*, artinya kedua anggota *class* tersebut bisa diakses oleh kode program di dalam *class* manapun. Seperti yang bisa Anda lihat di kode program 5-6 di atas, ketika kita membuat sebuah

objek bernama `mobil` di dalam `class Program`, kita bisa menginisialisasi *field* `Transmisi` dan juga bisa memanggil *method* `PindahTransmisi` melalui objek tersebut.

Untuk mengatasi masalah ini, ubah akses **public** untuk *field* `Transmisi` dan *method* `PindahTransmisi` menjadi **protected** sehingga kode program 6-6 menjadi seperti berikut ini.

`using System;`

```
namespace KodeProgram_6_7
{
    class KendaraanBermotor
    {
        protected string Transmisi;

        public KendaraanBermotor(string transmisi)
        {
            this.Transmisi = transmisi;
        }

        protected void PindahTransmisi()
        {
            Console.WriteLine("Rubah rasio putaran turbin");
        }
    }

    class Mobil : KendaraanBermotor
    {
        public Mobil(string transmisi): base (transmisi)
        {
        }
    }
}
```

```

public void JalanMaju()
{
    if (Transmisi == "Otomatis")
    {
        Console.WriteLine("Pindah tuas transmisi dari N ke D");
        Console.WriteLine("Injak pedal gas");
        PindahTransmisi();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Mobil mobil = new Mobil("Otomatis");

        // Kode di bawah menghasilkan eror
        mobil.Transmisi = "Otomatis";

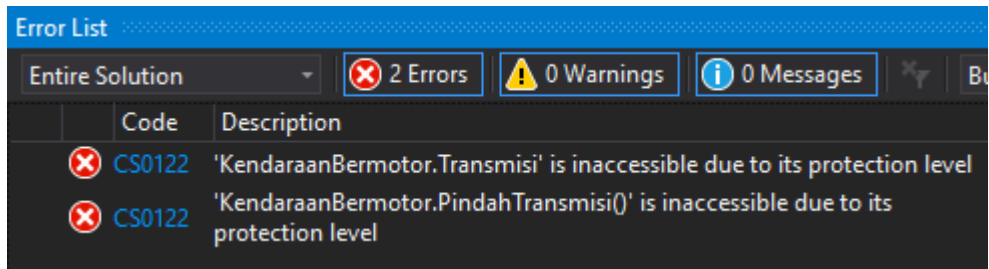
        mobil.JalanMaju();

        // Kode di bawah menghasilkan eror
        mobil.PindahTransmisi();

        Console.ReadLine();
    }
}

```

Apabila Anda menjalankan kode program di atas, Anda akan menemui eror yang menyatakan bahwa kedua anggota *class* tersebut tidak dapat diakses dari *class* Program seperti yang ditunjukkan pada Gambar 6.10.



Gambar 6.10

Bab 7

Mengenal Abstract Class Dan Method Overriding

Abstract class

Di bab 6, kita bisa membuat masing-masing objek dari *class* `Mobil` dan *class* `Motor` yang merupakan turunan dari *class* `KendaraanBermotor`. Namun, apakah masuk akal apabila Anda bisa membuat sebuah objek langsung dari *class* `KendaraanBermotor`?

“Kendaraan Bermotor” sendiri adalah sebuah konsep yang abstrak. Apabila Anda membuat sebuah objek dari *class* `KendaraanBermotor`, apakah objek tersebut merupakan sebuah mobil atau motor? Belum tentu keduanya, bukan?

Sama halnya ketika Anda mencoba membuat sebuah objek dari *class* `Mamalia`. Apakah objek yang Anda buat itu adalah seekor kucing, anjing, sapi, atau kambing? Tidak ada yang tahu.

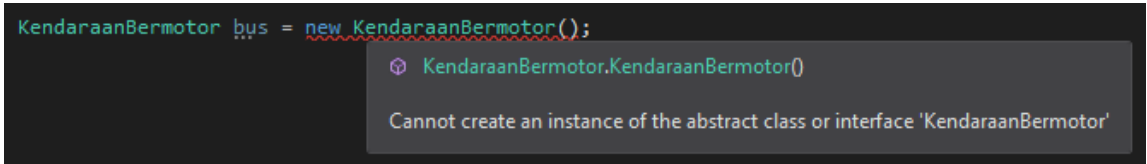
Tujuan kita membuat *class* `KendaraanBermotor` atau *class* `Mamalia` adalah hanya untuk mengumpulkan karakteristik dan perilaku yang bersifat umum dari *class* turunannya ke dalam satu wadah agar nantinya karakteristik dan

perilaku tersebut bisa diwariskan.

Dengan alasan seperti itu, lebih baik Anda mencegah instansiasi objek dari *class* `KendaraanBermotor` dengan cara mendeklarasikan *class* tersebut dengan kata kunci `abstract`. Sehingga, deklarasi *class* `KendaraanBermotor` tadi menjadi seperti berikut ini.

```
abstract class KendaraanBermotor
{
}
```

Apabila Anda mencoba untuk membuat sebuah objek langsung dari *class* `KendaraanBermotor` yang dideklarasikan dengan kata kunci `abstract`, *compiler* akan memberi Anda peringatan bahwa membuat sebuah objek (*instance*) dari *abstract class* itu tidak diperkenankan.



Gambar 7.1

Menyediakan method yang bisa ditimpa (overriding)

Pada *class* `KendaraanBermotor`, kita menyediakan sebuah *method* bernama `Jalan()` yang bisa digunakan oleh setiap objek dari semua *class* turunannya.

Namun, bagaimana jika prosedur untuk menjalankan sebuah mobil dan sebuah motor sedikit atau sama sekali berbeda?

Contohnya, untuk menjalankan sebuah mobil bertransmisi otomatis, kita perlu memindah tuas transmisi terlebih dahulu dari N ke D, lalu menginjak pedal gas. Sementara untuk menjalankan sebuah motor yang juga bertransmisi otomatis, kita hanya perlu memutar handle-gasnya saja. Disini kita bisa melihat dua prosedur yang sama sekali berbeda, bukan?

Lalu bagaimana agar `class Mobil` dan `class Motor` bisa mengimplementasikan `method Jalan()`-nya masing-masing meskipun `method` tersebut sudah disediakan oleh `class KendaraanBermotor`?

Solusinya adalah menjadikan `method Jalan()` pada `class KendaraanBermotor` menjadi *abstract method* atau *virtual method*.

Abstract method

Untuk membuat sebuah *abstract method*, Anda cukup menambahkan kata kunci `abstract` di depan nama sebuah *method* seperti berikut ini.

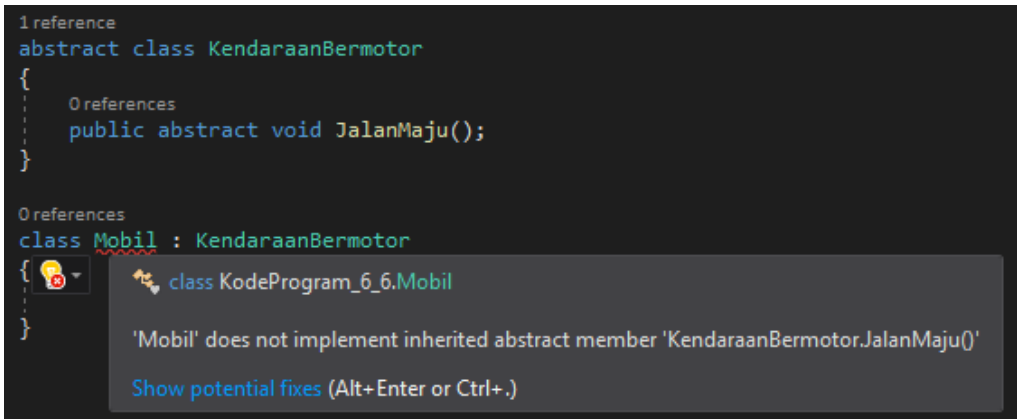
```
public abstract void JalanMaju();
```

Ada beberapa batasan ketika Anda memutuskan untuk membuat sebuah *abstract method*, yaitu:

- 1) Sebuah *abstract method* hanya bisa terdapat di dalam *abstract class* saja.
- 2) Sebuah *abstract method* tidak boleh memiliki implementasi apapun,

termasuk tanda kurung kurawal.

- 3) Sebuah *abstract method* harus diakhir dengan tanda titik dua (*semicolon*).
- 4) Sebuah *abstract method* harus diimplementasikan di dalam *class* turunannya. Apabila Anda tidak mengimplementasikan method tersebut, maka *compiler* akan memberi Anda peringatan seperti pada gambar 7.2.



Gambar 7.2

Virtual method

Sementara untuk membuat sebuah *virtual method*, Anda bisa menambahkan kata kunci **virtual** di depan nama sebuah *method* seperti berikut ini.

```
public virtual void JalanMaju() { }
```

Dengan menyediakan *virtual method* pada *base class*, berarti Anda

menyediakan mekanisme umum yang bisa ditimpa (*override*), ditambah, atau digunakan begitu saja oleh *class* turunannya.

Berbeda dengan *abstract method*, Anda perlu mengimplementasikan sebuah *virtual method* seperti halnya Anda mengimplementasikan sebuah *method* biasa (non-virtual). Selain itu, tidak seperti *abstract method* yang hanya bisa Anda deklarasikan di dalam *abstract class* saja, Anda bisa mendeklarasikan *virtual method* di dalam *abstract class* atau *class* biasa sekalipun.

Menggantikan method pada base class

Agar *class Mobil* dan *class Motor* dapat memiliki implementasi *method Jalan()*-nya masing-masing, selain membuat *method* tersebut menjadi *abstract method* ataupun *virtual method* pada *class KendaraanBermotor*, Anda juga perlu melakukan baik itu *method hiding* ataupun *method overriding*.

Dengan melakukan *method hiding*, artinya Anda **menyembunyikan** implementasi *method Jalan()* pada *class KendaraanBermotor* dari *class Mobil* dan *class Motor*, sehingga Anda dapat mendefinisikan ulang *method* tersebut pada *class Mobil* dan *class Motor*.

Sedangkan dengan melakukan *method overriding*, artinya Anda **mengganti** implementasi *method JalanMaju()* pada *class KendaraanBermotor* dengan masing-masing implementasi *method Jalan()* baik pada *class Mobil* maupun *class Motor*.

Method hiding

Dalam *method hiding*, Anda dapat menyembunyikan implementasi *method* pada *base class* dari *class* turunannya menggunakan kata kunci `new` seperti pada kode program berikut ini.

```
using System;

namespace KodeProgram_7_1
{
    abstract class KendaraanBermotor
    {
        public void Jalan()
        {
            Console.WriteLine("Berjalan maju");
        }
    }

    class Mobil : KendaraanBermotor
    {
        public new void Jalan()
        {
            Console.WriteLine("Menjalankan sebuah mobil:");
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
            Console.WriteLine("2. Injak pedal gas");
        }
    }

    class Motor : KendaraanBermotor
    {
        public new void Jalan()
        {
            Console.WriteLine("Menjalankan sebuah motor:");
            Console.WriteLine("1. Putar handle gas");
        }
    }
}
```

```

    }

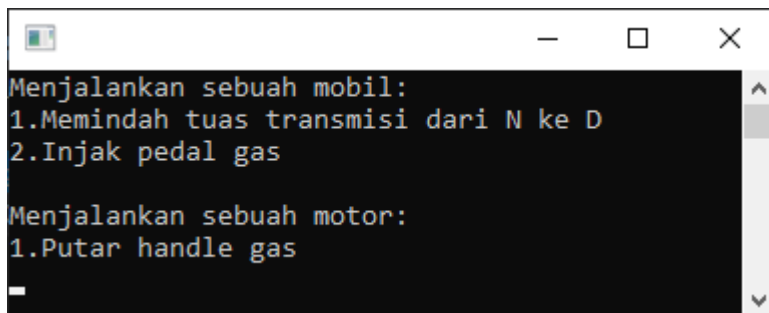
    class Program
    {
        static void Main(string[] args)
        {
            Mobil mobil = new Mobil();
            Motor motor = new Motor();

            mobil.Jalan();
            Console.WriteLine("");
            motor.Jalan();

            Console.ReadLine();
        }
    }
}

```

Kode program di atas akan menghasilkan keluaran seperti pada Gambar 7.3.



```

Menjalankan sebuah mobil:
1.Memindah tuas transmisi dari N ke D
2.Injak pedal gas

Menjalankan sebuah motor:
1.Putar handle gas

```


Gambar 7.3

Perhatikan keluaran dari kode program 7-1. Ketika kita memanggil *method* `Jalan()` melalui objek `mobil`, *method* yang dipanggil adalah *method* `Jalan()` pada *class* `Mobil`. Sementara itu, ketika kita memanggil *method*

`Jalan()` melalui objek `motor`, *method* `Jalan()` pada *class* `Motor` lah yang dipanggil.

Sebenarnya Anda bisa melakukan *method hiding* tanpa menambahkan kata kunci `new`. Kode program Anda akan tetap bisa dikompilasi, dijalankan tanpa adanya eror, dan menghasilkan keluaran yang sama seperti pada gambar 7-3.

Namun, Anda akan mendapatkan **peringatan** yang menyatakan bahwa *method* `Jalan()` pada *class* `Mobil` dan *class* `Motor` telah menyembunyikan *method* `Jalan()` yang diwariskan oleh *class* `KendaraanBermotor`, sehingga Anda diminta untuk menggunakan kata kunci `new` apabila Anda memang bermaksud untuk melakukan *method hiding*.

	Code	Description
	CS0108	'Motor.JalanMaju()' hides inherited member 'KendaraanBermotor.JalanMaju()'. Use the new keyword if hiding was intended.

Gambar 7.4

Method overriding

Untuk menggantikan implementasi *method* `Jalan()` pada *class* `KendaraanBermotor` dengan implementasi *method* `Jalan()` baik pada *class* `Mobil` maupun *class* `Motor`, Anda bisa menggunakan kata kunci `override` di antara *access modifier* dan nilai balik (*return value*) dari *method* `Jalan()` seperti berikut ini.

```
public override void JalanMaju()
{

}
```

Kode program di bawah ini mendemonstrasikan bagaimana cara mengimplementasikan sebuah *abstract method*.

```
using System;
```

```
namespace KodeProgram_7_2
{
    abstract class KendaraanBermotor
    {
        public abstract void Jalan();
    }

    class Mobil : KendaraanBermotor
    {
        public override void Jalan()
        {
            Console.WriteLine("Menjalankan sebuah mobil:");
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
            Console.WriteLine("2. Injak pedal gas");
        }
    }

    class Motor : KendaraanBermotor
    {
        public override void Jalan()
        {
            Console.WriteLine("Menjalankan sebuah motor:");
            Console.WriteLine("1. Putar handle gas");
        }
    }
}
```

```

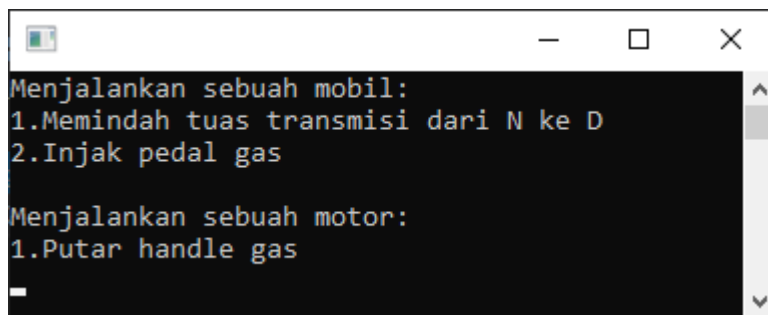
class Program
{
    static void Main(string[] args)
    {
        Mobil mobil = new Mobil();
        Motor motor = new Motor();

        mobil.JalanMaju();
        Console.WriteLine("");
        motor.JalanMaju();

        Console.ReadLine();
    }
}

```

Kode program di atas akan menghasilkan keluaran yang sama persis dengan gambar 7.3.



Gambar 7.5

Anda juga bisa melakukan *method overriding* dengan menggunakan *virtual method* disamping menggunakan *abstract method*. Apabila Anda mengganti implementasi *method* `Jalan()` pada class `KendaraanBermotor` yang dideklarasikan dengan kata kunci `virtual`, Anda juga akan mendapatkan

keluaran yang sama seperti pada gambar 7.3 dan Gambar 7.5

```
using System;
```

```
namespace KodeProgram_7_3
```

```
{  
    abstract class KendaraanBermotor  
    {  
        public virtual void Jalan()  
        {  
            Console.WriteLine("Berjalan maju");  
        }  
    }  
  
    class Mobil : KendaraanBermotor  
    {  
        public override void Jalan()  
        {  
            Console.WriteLine("Menjalankan sebuah mobil:");  
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");  
            Console.WriteLine("2. Injak pedal gas");  
        }  
    }  
  
    class Motor : KendaraanBermotor  
    {  
        public override void Jalan()  
        {  
            Console.WriteLine("Menjalankan sebuah motor:");  
            Console.WriteLine("1. Putar handle gas");  
        }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)
```



```

{
    Mobil mobil = new Mobil();
    Motor motor = new Motor();

    mobil.Jalan();
    Console.WriteLine("");
    motor.Jalan();

    Console.ReadLine();
}
}
}

```

Coba jalankan kode program 7-3 dan bandingkan keluarannya dengan kode program 7-1 dan 7-2.

Sampai di sini mungkin Anda bertanya, jika menerapkan *method hiding* dan *method overriding* menghasilkan keluaran yang sama, maka kapan sebaiknya kita menggunakan *method hiding* atau *method overriding*?

Anda akan menemukan jawabannya ketika mempelajari tentang polimorfisme di bab selanjutnya.

Bab 8

Perilaku Polimorfik Dan Ekstensibilitas Software

Tujuan utama dari teknik pewarisan sebenarnya adalah untuk mendapatkan perilaku polimorfik pada sebuah objek. Perilaku polimorfik ini bisa Anda dapatkan dengan cara membuat objek-objek dari sebuah *derived class* seolah-olah adalah objek dari *base class*-nya.

Pada Bab ini, Anda akan mempelajari bagaimana menggunakan teknik pewarisan untuk mendapatkan perilaku polimorfik pada sebuah objek.

Setelah selesai mempelajari bab ini, Anda juga akan memahami bahwa dengan menerapkan konsep polimorfisme, Anda bisa mewujudkan ekstensibilitas pada aplikasi yang Anda kembangkan. Artinya, Anda bisa terus-menerus melakukan penambahan fungsi pada aplikasi Anda tanpa harus merubah kode program yang sudah ada sebelumnya.

Tipe Objek

Masih ingat Nyanyan kucing Anda yang kita bahas di Bab 6? Katakan Anda mendapat pertanyaan seperti “Apakah Nyanyan?”, Anda mungkin akan memiliki bermacam-macam jawaban tergantung dari sudut pandang Anda

melihat Nyanyan. Jawaban Anda bisa dari yang paling spesifik sampai yang paling umum, seperti:

- Nyanyan **adalah** seekor kucing
- Nyanyan **adalah** seekor mamalia
- Nyanyan **adalah** seekor binatang

Jawaban-jawaban Anda di atas semuanya valid, dan tentunya didasarkan pada sebuah hirarki yang menyatakan bahwa kucing adalah mamalia dan mamalia adalah binatang.

Dengan demikian, apabila kita melihat kembali kode program kita sebelumnya di mana *class Mobil* dan *class Motor* merupakan *class* turunan dari *class KendaraanBermotor*, maka kita bisa menyatakan bahwa baik objek dari *class Mobil* maupun *class Motor* adalah dari tipe *class KendaraanBermotor*.

Konsekuensinya adalah, Anda bisa membuat kedua objek tersebut seolah-olah adalah objek dari tipe *KendaraanBermotor* seperti pada potongan kode program di bawah ini:

```
KendaraanBermotor mobil = new Mobil();
KendaraanBermotor motor = new Motor();
```

Referensi ke tipe *Mobil* dan *Motor* di atas masing-masing bukan lagi bertipe *Mobil* dan *Motor*, melainkan bertipe *KendaraanBermotor*.

Menerapkan polimorfisme

Memperlakukan sebuah objek dari *derived class* seolah-olah adalah objek dari *base class*-nya merupakan dasar dari penerapan polimorfisme.

Untuk lebih memahaminya, coba perhatikan kode program 8-1 berikut ini.

```
using System;

namespace KodeProgram_8_1
{
    abstract class KendaraanBermotor
    {
        public virtual void Jalankan()
        {
            Console.WriteLine("Berjalan maju");
        }
    }

    class Mobil : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah mobil:");
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
            Console.WriteLine("2. Injak pedal gas");
        }
    }

    class Motor : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah motor:");
            Console.WriteLine("1. Putar handle gas");
        }
    }
}
```

```

    }

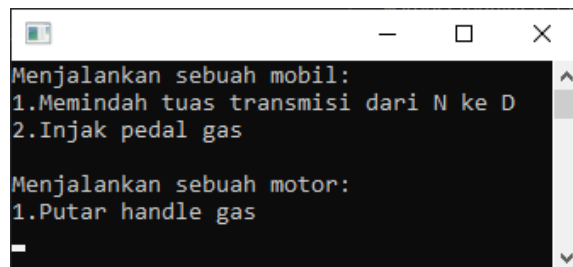
    class Program
    {
        static void Main(string[] args)
        {
            KendaraanBermotor mobil = new Mobil();
            KendaraanBermotor motor = new Motor();

            mobil.Jalankan();
            Console.WriteLine("");
            motor.Jalankan();

            Console.ReadLine();
        }
    }
}

```

Jika Anda perhatikan, kode program di atas mirip dengan kode program 7-3 di Bab 7. Bedanya, kali ini kita membuat objek mobil dan motor dari tipe `KendaraanBermotor`. Kode program di atas pun jika Anda jalankan akan menghasilkan keluaran yang sama seperti kode program 7-3.



Gambar 8.1

Mengapa keluarannya bisa sama, padahal kali ini objek yang kita buat adalah

dari tipe `KendaraanBermotor`?

Inilah yang disebut dengan polimorfisme. Ketika sebuah objek dari tipe `KendaraanBermotor` diberi referensi ke tipe `Mobil`, maka objek tersebut akan berperilaku seperti objek bertipe `Mobil`. Begitu pula ketika objek tersebut diberi referensi ke tipe `Motor`, maka dia akan berperilaku seperti objek bertipe `Motor`.

Perhatikan kembali bahwa perilaku polimorfik di atas bisa kita dapatkan dengan memanfaatkan teknik pewarisan dan juga *method overriding*. Apa yang terjadi jika Anda memilih untuk menerapkan *method hiding*, daripada *method overriding*? Apakah Anda akan mendapatkan keluaran yang sama? Untuk menjawab pertanyaan tersebut, coba perhatikan kode program 8-2 berikut ini.

```
using System;

namespace KodeProgram_8_2
{
    abstract class KendaraanBermotor
    {
        public void Jalankan()
        {
            Console.WriteLine("Berjalan maju");
        }
    }

    class Mobil : KendaraanBermotor
    {
        public new void Jalankan()
        {
```

```

        Console.WriteLine("Menjalankan sebuah mobil:");
        Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
        Console.WriteLine("2. Injak pedal gas");
    }
}

class Motor : KendaraanBermotor
{
    public new void Jalankan()
    {
        Console.WriteLine("Menjalankan sebuah motor:");
        Console.WriteLine("1. Putar handle gas");
    }
}

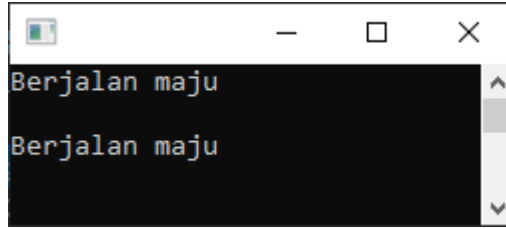
class Program
{
    static void Main(string[] args)
    {
        KendaraanBermotor mobil = new Mobil();
        KendaraanBermotor motor = new Motor();

        mobil.Jalankan();
        Console.WriteLine("");
        motor.Jalankan();

        Console.ReadLine();
    }
}

```

Kode program di atas akan menghasilkan keluaran seperti pada gambar 8.2.



Gambar 8.2

Sepertinya kali ini, keluarannya tidak seperti yang kita harapkan. Nampaknya yang dieksekusi adalah *method* `Jalankan()` pada *class* `KendaraanBermotor`.

Tidak ada perilaku polimorfik di sini. Objek dari tipe `KendaraanBermotor` tetap akan mengeksekusi *method* pada *class*-nya sendiri, tidak peduli referensi tipe apapun yang diberikan kepadanya.

Sampai di sini jelas bahwa *method hiding* tidak bisa Anda manfaatkan untuk mendapatkan perilaku polimorfik. Teknik ini hanya digunakan untuk menyembunyikan sebuah *method* pada *base class* dari *class* turunannya.

Dengan *method hiding*, ketika Anda membuat sebuah objek dari tipe `KendaraanBermotor`, tidak peduli referensi apapun baik ke tipe `Mobil` maupun `Motor` yang diberikan ke objek tersebut, dia tetap akan tahu bahwa pada *class* `KendaraanBermotor` terdapat sebuah *method* yang bisa dieksekusi. Mengapa? Karena objek tersebut sejatinya adalah objek dari tipe `KendaraanBermotor`.

Interaksi antar objek

Untuk menjalankan sebuah mobil atau motor, biasanya dibutuhkan seorang pengemudi. Selain itu, agar seorang pengemudi bisa berinteraksi dengan sebuah mobil atau motor, dibutuhkan sebuah antarmuka.

Contohnya, untuk mengendarai sebuah mobil, pengemudi memerlukan antarmuka seperti setir kemudi, pedal gas, pedal rem, dan gagang transmisi. Untuk mengendarai sebuah motor, dia memerlukan antarmuka yang berbeda, yaitu stang kemudi dan *handle* gas.

Dengan demikian, apabila si pengemudi tahu bagaimana berinteraksi dengan antarmuka sebuah mobil, maka dia pastinya bisa mengendarai sebuah mobil. Begitupun jika dia tahu bagaimana berinteraksi dengan antarmuka sebuah motor, maka dia juga bisa mengendarai sebuah motor.

Agar Anda bisa lebih memahami bagaimana interaksi antar objek di C#, mari kita kembali bereksperimen dengan kode program. Kita akan memanfaatkan *class* `Mobil` dan *class* `Motor` yang sebelumnya telah kita definisikan di kode program 8-2. Kemudian, kita akan menambahkan satu *class* yang diberi nama `Pengemudi`.

Kita akan melihat sejauh mana objek dari *class* `Pengemudi` ini bisa berinteraksi baik dengan objek dari *class* `Mobil` maupun *class* `Motor`.

```
using System;
```

```
namespace KodeProgram_8_3
```

```
{
    abstract class KendaraanBermotor
    {
        protected virtual void Jalankan()
        {
            Console.WriteLine("Berjalan maju");
        }
    }

    class Mobil : KendaraanBermotor
    {
        protected override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah mobil:");
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
            Console.WriteLine("2. Injak pedal gas");
        }
    }

    class Motor : KendaraanBermotor
    {
        protected override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah motor:");
            Console.WriteLine("1. Putar handle gas");
        }
    }

    class Pengemudi
    {
        public void Mengemudi(Mobil mobil)
        {
            mobil.Jalankan();
        }
    }

    class Program
```

```

{
    static void Main(string[] args)
    {
        Mobil mobil = new Mobil();
        Motor motor = new Motor();

        Pengemudi pengemudi = new Pengemudi();

        pengemudi.Mengemudi(mobil);

        // baris kode program berikut ini menghasilkan error
        // pengemudi hanya mengerti tipe Mobil saja
        pengemudi.Mengemudi(motor);

        Console.ReadLine();
    }
}

```

Pada kode program di atas, tipe `Mobil` pada *method* `Mengemudi(Mobil mobil)` merupakan antarmuka objek mobil yang kita berikan pada objek `pengemudi`. Konsekuensinya, objek `pengemudi` ini hanya mengerti antarmuka objek mobil saja. Sehingga, ketika kita memberikan objek `motor` sebagai argumen untuk *method* `Mengemudi()`, kode program Anda akan menghasilkan error.

Lalu bagaimana membuat objek `pengemudi` bisa berinteraksi baik dengan objek mobil maupun objek motor?

Jawabannya, Anda harus menyediakan antarmuka yang memungkinkan Anda mendapatkan perilaku polimorfik.

Antarmuka polimorfik

Anda sudah mempelajari apa itu polimorfisme dan bagaimana mendapatkan perilaku polimorfik pada sebuah objek. Tapi, apa itu antarmuka polimorfik?

Pada kode program 8-3, Anda bisa melihat bahwa objek pengemudi gagal mengemudikan objek motor karena objek pengemudi hanya mengerti antarmuka dari objek mobil.

Agar objek pengemudi bisa mengendarai kedua objek tersebut, maka kita seharusnya menyediakan antarmuka generik (umum) dari kedua objek tersebut untuknya.

Apa bentuk generik dari mobil dan motor? Betul, kendaraan bermotor.

Daripada mengatakan secara spesifik bahwa si pengemudi bisa mengemudikan sebuah mobil atau motor, lebih baik kita mengatakan bahwa si pengemudi bisa mengendarai sebuah kendaraan bermotor.

Jika kita menyebut antarmuka sebuah kendaraan bermotor, maka itu bisa saja diartikan sebagai antarmuka sebuah mobil, motor, bus, ataupun berbagai kendaraan bermotor lainnya. Inilah yang disebut dengan antarmuka polimorfik. Sebuah antarmuka yang bisa diinterpretasikan atau diartikan ke berbagai macam antarmuka.

Agar Anda bisa lebih memahaminya, perhatikan kode program di bawah ini.

```
using System;
```

```

namespace KodeProgram_8_4
{
    abstract class KendaraanBermotor
    {
        public virtual void Jalankan()
        {
            Console.WriteLine("Berjalan maju");
        }
    }

    class Mobil : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah mobil:");
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
            Console.WriteLine("2. Injak pedal gas");
        }
    }

    class Motor : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah motor:");
            Console.WriteLine("1. Putar handle gas");
        }
    }

    class Pengemudi
    {
        public void Mengemudi (KendaraanBermotor kb)
        {
            kb.Jalankan();
        }
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        KendaraanBermotor mobil = new Mobil();
        KendaraanBermotor motor = new Motor();

        Pengemudi pengemudi = new Pengemudi();

        pengemudi.Mengemudi(mobil);
        Console.WriteLine("");
        pengemudi.Mengemudi(motor);

        Console.ReadLine();
    }
}

```

Kali ini, daripada menyediakan tipe `Mobil` sebagai parameter pada *method* `Mengemudi(Mobil mobil)`, kita menggunakan tipe `KendaraanBermotor` sebagai parameter. Tipe `KendaraanBermotor` inilah yang kita jadikan antarmuka agar objek `pengemudi` bisa berinteraksi dengan objek apapun yang masih termasuk dari tipe `KendaraanBermotor`.

Seperti yang bisa Anda lihat pada kode program 8-4 di atas, ketika *method* `Mengemudi()` milik objek `pengemudi` diberi objek `motor` sebagai argumen, kali ini kode program kita tidak akan menghasilkan error seperti sebelumnya.

Ekstensibilitas

Arsitektur sebuah software bisa dikatakan baik, salah satunya, jika bisa

beradaptasi dengan perubahan atau penambahan fitur tanpa harus merubah dan mengompilasi ulang kode program yang sudah ada. Hal ini dinamakan dengan *open closed principle (OCP)* yang menyatakan bahwa sebuah entitas software (*class* atau *method*) harus dibuat terbuka agar bisa diperluas fungsinya namun tertutup dari segala bentuk modifikasi.

Dengan menyediakan antarmuka polimorfik untuk objek pengemudi, artinya Anda bisa memperluas fungsi dari objek tersebut. Tidak hanya bisa mengendarai mobil atau motor saja, objek pengemudi ini juga kedepannya dimungkinkan untuk mengendarai berbagai objek yang masih termasuk dari tipe `KendaraanBermotor`.

Selain itu, kedepannya Anda bisa menambah lebih banyak lagi objek dari tipe `KendaraanBermotor` seperti Bus dan truk. Tanpa harus menyentuh *class* `Pengemudi`, objek pengemudi tetap bisa mengendarai objek-objek baru tersebut.

Mari kita kembali bereksperimen. Kali ini, kita akan membuat sebuah *class* baru bernama `Bus`. Selain itu, karena kendaraan bermotor adalah konsep yang abstrak, dan tentunya cukup sulit untuk mendefinisikan sebuah mekanisme dari sesuatu yang abstrak, kita akan mengganti *virtual method* pada *class* `KendaraanBermotor` menjadi *abstract method*.

Dengan demikian, kini *class* `KendaraanBermotor` merupakan sebuah *class* yang benar-benar abstrak, tanpa implementasi.

```
using System;

namespace KodeProgram_8_5
{
    abstract class KendaraanBermotor
    {
        public abstract void Jalankan();
    }

    class Mobil : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah mobil:");
            Console.WriteLine("1. Memindah tuas transmisi dari N ke D");
            Console.WriteLine("2. Injak pedal gas");
        }
    }

    class Motor : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah motor:");
            Console.WriteLine("1. Putar handle gas");
        }
    }

    class Bus : KendaraanBermotor
    {
        public override void Jalankan()
        {
            Console.WriteLine("Menjalankan sebuah bus:");
            Console.WriteLine("1. Memindah tuas transmisi");
            Console.WriteLine("2. Injak kopling");
            Console.WriteLine("3. Injak pedal gas, lepas pedal kopling pelan-  
pelan");
        }
    }
}
```



```

    }
}

class Pengemudi
{
    public void Mengemudi (KendaraanBermotor kb)
    {
        kb.Jalankan();
    }
}

class Program
{
    static void Main(string[] args)
    {
        KendaraanBermotor mobil = new Mobil();
        KendaraanBermotor motor = new Motor();
        KendaraanBermotor bus = new Bus();

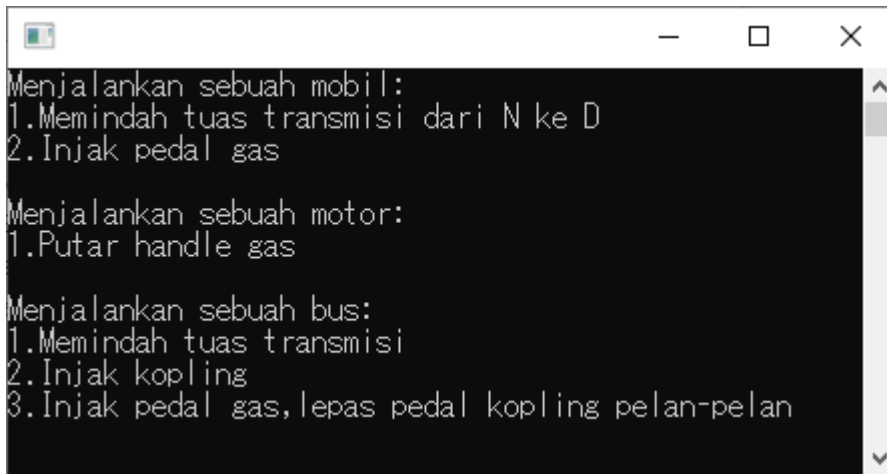
        Pengemudi pengemudi = new Pengemudi();

        pengemudi.Mengemudi(mobil);
        Console.WriteLine("");
        pengemudi.Mengemudi(motor);
        Console.WriteLine("");
        pengemudi.Mengemudi(bus);

        Console.ReadLine();
    }
}

```

Kode program di atas akan menampilkan keluaran seperti pada Gambar 8.3.



Gambar 8.3

Menarik, bukan? Ketika kita menambahkan sebuah tipe baru dan membuat objek dari tipe tersebut, tanpa harus menyentuh *class* `Pengemudi` pun, objek dari *class* `Pengemudi` masih bisa berinteraksi dengan objek tersebut.

Dengan membuatnya tertutup dari segala bentuk modifikasi, namun terbuka untuk dapat diperluas fungsinya, artinya kita juga telah berhasil menerapkan prinsip OCP pada *class* `Pengemudi`.

Keterbatasan abstract class sebagai antarmuka polimorfik

Meskipun menggunakan *abstract class* bisa membantu Anda untuk mendapatkan perilaku polimorfik, Anda tetap harus cermat ketika memutuskan untuk menggunakannya.

Kedepannya, ada kemungkinan Anda akan menemui beberapa masalah

ketika kurang cermat dalam menerapkan sebuah hirarki dengan *abstract class*. Hal ini dikarenakan *abstract class* memiliki beberapa keterbatasan pada penerapannya.

Paling tidak, ada dua keterbatasan yang bisa Anda temui pada saat menggunakan *abstract class* sebagai antarmuka polimorfik.

Yang pertama, katakanlah Anda memiliki *abstract class* yang mendefinisikan sebuah kendaraan bermotor seperti pada kode program di bawah ini.

```
abstract class KendaraanBermotor
{
    // Setiap class turunannya harus mengimplementasikan method ini
    public abstract void IsiBahanBakar();
    public abstract void Jalankan();
}
```

Kemudian, Anda juga memiliki beberapa *class* seperti *Mobil*, *Motor*, *Bus*, *Truk*, *Pesawat*, dan *MobilListrik*. *Class-class* tersebut Anda turunkan dari *abstract class KendaraanBermotor*.

Anda melihat ada masalah di sini?

Kalau Anda mengatakan bahwa *MobilListrik* tidak bisa menerapkan *method* *IsiBahanBakar()*, berarti Anda cukup jeli. Sebuah *MobilListrik* memang bisa dikategorikan sebagai *KendaraanBermotor*, namun *MobilListrik* digerakkan oleh motor listrik. Motor listrik butuh energi listrik, bukan bahan bakar.

Bagaimana dengan solusi di bawah ini?

```
abstract class KendaraanBermotor
{
    public abstract void Jalankan();
}

abstract class TipeYangBisaDiisiBahanBakar
{
    public abstract void IsiBahanBakar();
}

abstract class TipeYangBisaDichargeListrik
{
    public abstract void ChargeBaterai();
}

// Mewarisi class KendaraanBermotor dan
// class TipeYangBisaDiisiBahanBakar secara bersamaan
class MobilBerbahanBakar : KendaraanBermotor, TipeYangBisaDiisiBahanBakar
{
    //implementasi method Jalankan() pada class KendaraanBermotor
    public override void Jalankan()
    {
        // instruksi untuk berjalan
    }

    //implementasi method IsiBahanBakar() pada class TipeYangBisaDiisiBahanBakar
    public override void IsiBahanBakar()
    {
        // instruksi untuk mengisi bahan bakar
    }
}

// Mewarisi class KendaraanBermotor dan
// class TipeYangBisaDichargeListrik secara bersamaan
class MobilListrik : KendaraanBermotor, TipeYangBisaDichargeListrik
{
```

```
//implementasi method Jalankan() pada class KendaraanBermotor
public override void Jalankan()
{
    // instruksi untuk berjalan
}

//implementasi method ChargeBaterai() pada class TipeYangBisaDichargeListrik
public override void ChargeBaterai()
{
    // instruksi untuk mengisi bahan bakar
}
}
```

Sayang sekali, terdapat satu masalah pada solusi di atas.

Pewarisan berganda (*multiple inheritance*) tidak dimungkinkan di pemrograman C#. Class `MobilBerbahanBakar` tidak bisa mewarisi class `KendaraanBermotor` dan class `TipeYangBisaDiisiBahanBakar` secara bersamaan. Kode program Anda di atas akan menghasilkan eror. Begitu pula dengan class `MobilListrik`.

Keterbatasan *abstract class* yang selanjutnya adalah **sebuah *class* harus merupakan turunan dari *abstract class* tersebut agar bisa mengimplementasikan mekanismenya.**

Coba kita perhatikan kembali class `TipeYangBisaDichargeListrik` dari kode program sebelumnya.

```
abstract class TipeYangBisaDichargeListrik
{
    // Sebuah class harus merupakan class turunan dari class MotorListrik
    // agar bisa menerapkan fungsionalitas dari method ini
}
```

```
    public abstract void ChargeBaterai();  
}
```

Mobil listrik **adalah** sebuah kendaraan, sedangkan kamera, laptop, dan *smartphone* **adalah** sebuah perangkat *mobile*. Meskipun objek-objek tersebut berasal dari hirarki objek yang berbeda, keduanya sama-sama merupakan tipe objek yang baterainya bisa diisi kembali.

Masalahnya, Anda sudah tahu bahwa C# tidak mendukung pewarisan berganda.

Lalu apa solusi dari permasalahan di atas? Jawabannya adalah menggunakan interface yang akan saya bahas di bab selanjutnya.

Bab 9

Interface

Pada bab sebelumnya Anda sudah melihat bagaimana menggunakan *abstract class* sebagai antarmuka polimorfik memungkinkan Anda untuk memperluas fungsionalitas atau fitur dari aplikasi Anda. Hanya saja kekurangannya adalah, setiap *class* yang Anda definisikan harus berada dalam sebuah hirarki pewarisan agar dapat diakses melalui *abstract class*-nya.

Pada skenario di mana Anda dituntut untuk memiliki aplikasi software dengan fleksibilitas yang tinggi, Anda bisa kehilangan fleksibilitas sehingga terpaksa menulis ulang kode program Anda (melakukan *refactoring*) agar dapat memenuhi tuntutan yang ada.

Beruntung, C# menyediakan *interface* yang bisa Anda manfaatkan sebagai antarmuka polimorfik.

Pada bab ini, Anda akan mempelajari tentang apa itu *interface* sebagai bekal untuk memecahkan permasalahan yang muncul di bab 8.

Interface dan abstraksi

Dalam Bahasa Indonesia, *interface* sendiri berarti antarmuka.

Ketika Anda berinteraksi dengan sebuah software, Anda berinteraksi dengan sebuah antarmuka, yaitu tampilan software di layar monitor.

Ketika Anda berinteraksi dengan sebuah mobil, Anda juga berinteraksi dengan beberapa antarmuka, yaitu setir kemudi, pedal gas dan rem, serta *handle* transmisi.

Berinteraksi melalui sebuah antarmuka memungkinkan Anda berinteraksi dengan suatu sistem atau objek pada level abstraksi. Artinya, Anda tidak perlu mengkhawatirkan bagaimana sistem atau objek tersebut bekerja secara detail.

Anda tidak perlu tahu bagaimana mesin mobil bekerja untuk sekedar bisa mengemudikannya. Waktu belajar mengemudi, Anda belajar bagaimana menggunakan antar muka yang disediakan untuk Anda. Bukan belajar bagaimana mesin mobil bekerja, bukan?

Mendefinisikan interface

Dalam pemrograman C#, Anda bisa mendeklarasikan sebuah *interface* seperti pada potongan kode program di bawah ini.

```
public interface INamaInterface
{
    void NamaMethod();
}
```

Pada potongan kode program di atas, `INamaInterface` adalah *identifier* atau

nama *interface* yang Anda deklarasikan. Sesuai dengan konvensi, nama sebuah *interface* selalu di mulai dengan huruf I besar.

Sama seperti sebuah *abstract class*, Anda tidak dapat membuat objek dari sebuah *interface*.

```
class Program
{
    static void Main(string[] args)
    {
        // Kode program di bawah ini akan menghasilkan eror
        INamaInterface objInterface = new INamaInterface();
    }
}
```

Selain itu, Anda juga tidak dapat mengimplementasikan sebuah *method* di dalam sebuah *interface* sama seperti saat mendeklarasikan sebuah *method* dengan kata kunci **abstract**. Potongan kode program di bawah ini akan menghasilkan eror.

```
public interface INamaInterface
{
    // menghasilkan eror
    void NamaMethod()
    {
        // implementasi method
    }
}
```

Berbeda dengan *abstract class*, Anda tidak bisa mendeklarasikan *field* di dalam sebuah *interface*.

```
public interface INamaInterface
{
    // menghasilkan eror
    public int sebuahField;
    void NamaMethod();
}
```

Namun demikian, Anda bisa mendeklarasikan beberapa properti tanpa menggunakan *access modifier* (`public`, `private`, `protected`, dan lain-lain) di dalam sebuah *interface* seperti pada potongan kode program di bawah ini.

```
public interface INamaInterface
{
    int ReadWriteProperty { get; set; }
    int ReadOnlyProperty { get; }
    int WriteOnlyProperty { set; }
}
```

Interface sebagai kontrak

Sebuah mobil biasanya menyediakan pedal gas sebagai antarmuka untuk menjalankan mesinnya. Karena semua mobil menggunakan pedal gas sebagai antarmuka untuk menjalankan mesinnya, sebagai pengguna, kita tahu dengan menginjak pedal gas, mesin mobil akan beroperasi.

Tidak masuk akal apabila ada sebuah mobil menggunakan pedal gas sebagai antarmuka untuk melakukan operasi yang berbeda. Misalnya, ketika kita menginjak pedal gas, justru *wiper* mobil yang bekerja.

Oleh karena itu, pedal gas ini seolah-olah adalah **sebuah kontrak untuk**

melakukan operasi tertentu. Artinya, setiap mobil yang menggunakan antarmuka pedal gas, harus mengimplementasikan operasi yang sama untuk antarmuka tersebut, yaitu untuk menjalankan mesinnya.

Implementasi interface pada sebuah class

Agar nantinya kita bisa berinteraksi dengan objek dari sebuah *class* melalui sebuah *interface*, kita perlu mengimplementasikan *interface* pada *class*-nya terlebih dulu.

Langkah-langkah berikut ini adalah cara mengimplementasikan sebuah *interface* pada sebuah *class*. Saya akan kembali menggunakan *class* `Mobil` sebagai contoh.

Pertama, kita definisikan sebuah *interface* bernama `IKendaraan` terlebih dulu. Nantinya, kita akan menggunakannya sebagai kontrak yang menyatakan bahwa setiap objek dari *class* dengan `IKendaraan` sebagai antarmuka, harus bisa dijalankan (wajib mengimplementasikan *method* `Jalankan()`).

```
public interface IKendaraan
{
    // setiap class dengan IKendaraan sebagai antar muka
    // harus mengimplementasikan method Jalankan()
    void Jalankan();
}
```

Selanjutnya, buat sebuah *class* bernama `Mobil`.

```
public class Mobil  
{  
  
}
```

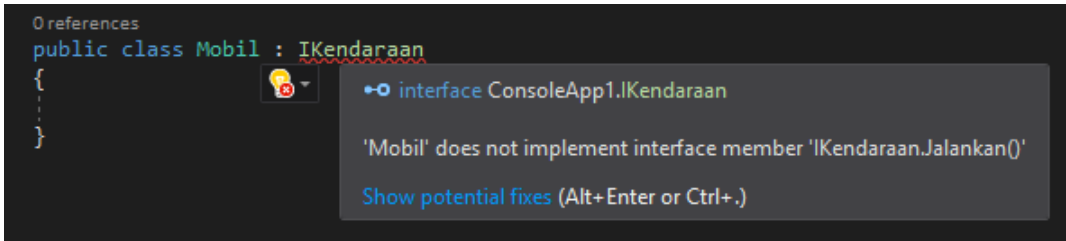
Setelah itu, kita bisa membuat `class Mobil` mengimplementasikan `IKendaraan` dengan cara seperti pada kode program di bawah ini.

```
public class Mobil : IKendaraan  
{  
  
}
```

Sebelumnya, dengan melihat sebuah mobil dari prespektif pewarisan, kita bisa mengatakan bahwa sebuah mobil **adalah** sebuah kendaraan. Oleh karena itu, sebuah mobil **mewarisi seluruh karakteristik dan perilaku umum** yang ada pada sebuah kendaraan.

Berbeda dengan pewarisan, kode program di atas menyatakan bahwa `class Mobil` **bersedia mengimplementasikan** seluruh *method* yang **telah ditentukan** di dalam `IKendaraan`.

Karena `IKendaraan` adalah kontrak yang harus dipatuhi oleh `class Mobil`, mau tidak mau, `class Mobil` wajib mengimplementasikan *method* `Jalankan()`. Jika tidak, *compiler* akan memberi Anda pesan error seperti yang ditampilkan di Gambar 9.1.



Gambar 9.1

Untuk mengimplementasikan *method* `Jalankan()` yang telah ditentukan oleh `IKendaraan`, kita hanya perlu membuat sebuah *method* dengan nama yang sama di dalam *class* `Mobil`. Perhatikan juga bahwa kita tidak perlu menggunakan kata kunci `override` seperti sebelumnya.

```
public class Mobil : IKendaraan
{
    public void Jalankan()
    {
        Console.WriteLine("Injak pedal gas");
    }
}
```

Kini, Anda bisa berinteraksi dengan *class* `Mobil` melalui antar muka `IKendaraan` karena *class* `Mobil` sudah memenuhi kontrak yang ditetapkan oleh `IKendaraan`.

Antarmuka polimorfik menggunakan interface

Anda sudah memahami bahwasanya menggunakan *abstract class* sebagai antarmuka polimorfik memiliki beberapa keterbatasan. Seperti kasus mobil dan smartphone di bab 8.

Dengan menggunakan *interface*, Anda bisa menyelesaikan permasalahan tersebut. Perhatikan contoh kode program berikut ini.

```
using System;

namespace KodeProgram_9_1
{
    public interface IBisaDichargeListrik
    {
        string ChargeBaterai();
    }

    abstract class KendaraanBermotor
    {
        public abstract string Jalankan();
    }

    abstract class PerangkatMobile
    {
        public abstract string Operasikan();
    }

    class Mobil : KendaraanBermotor, IBisaDichargeListrik
    {
        public string ChargeBaterai()
        {
            return "Mengisi Baterai Mobil Listrik";
        }

        public override string Jalankan()
        {
            return "Menjalankan Mobil";
        }
    }

    class Smartphone : PerangkatMobile, IBisaDichargeListrik
```

```

{
    public string ChargeBaterai()
    {
        return "Mengisi Baterai Smartphone";
    }

    public override string Operasikan()
    {
        return "Mengoperasikan Smartphone";
    }
}

class Program
{
    static void Main(string[] args)
    {
        IBisaDichargeListrik mobil = new Mobil();
        IBisaDichargeListrik smartphone = new Smartphone();

        Console.WriteLine(Charger(mobil));
        Console.WriteLine(Charger(smartphone));
        Console.ReadLine();
    }

    private static string Charger(IBisaDichargeListrik ch)
    {
        return ch.ChargeBaterai();
    }
}

```

Kode program di atas menghasilkan keluaran seperti yang ditampilkan pada Gambar 9.2.



Gambar 9.2

Kita bisa melihat bahwa `Mobil` dan `Smartphone` merupakan *class* dengan hirarki yang berbeda. `Mobil` adalah `KendaraanBermotor`, sedangkan `Smartphone` adalah `PerangkatMobile`. Namun demikian, baik objek dari *class* `Mobil` maupun dari *class* `Smartphone` bisa kita berikan sebagai argumen pada *method* `Charger()` karena keduanya mengimplementasikan *interface* `IBisaDichargeListrik`.

Mengimplementasikan lebih dari satu interface

“Bahasa pemrograman C# tidak mendukung pewarisan berganda pada sebuah class. Class C tidak bisa mewarisi class A dan class B sekaligus. Namun dengan menggunakan interface, kita bisa seolah-olah melakukan pewarisan berganda pada sebuah class. Class C bisa mewarisi interface A dan interface B sekaligus.”

Begitu kira-kira yang sering saya baca ketika mempelajari tentang *interface*. Menurut saya, pernyataan yang saya cetak tebal seperti di atas adalah pernyataan yang **menyesatkan**.

Sebuah *class* tidak bisa mewarisi *interface*. Sebuah mobil **bukanlah** sebuah pedal gas! Sebuah mobil hanya mengimplementasikan kontrak dari pedal

gas yang menyatakan bahwa dengan menginjak pedal gas, mobil bisa berakselerasi.

Dalam pemrograman C#, sebuah *class* dimungkinkan untuk mengimplementasikan lebih dari satu *interface*. Seperti halnya sebuah mobil bisa mengimplementasikan beberapa antarmuka seperti pedal gas/rem, setir kemudi, dan tuas transmisi.

Lalu bagaimana mengimplementasikan lebih dari satu interface pada sebuah *class*? Perhatikan kode program di bawah ini.

```
namespace KodeProgram_9_2
{
    public interface IBisaDichargeListrik
    {
        string ChargeBaterai();
    }

    public interface IBisaDiisiBahanBakar
    {
        string IsiBahanBakar();
    }

    class MobilHybrid : IBisaDiisiBahanBakar, IBisaDichargeListrik
    {
        public string ChargeBaterai()
        {
            return "Mengisi Baterai";
        }

        public string IsiBahanBakar()
        {
            return "Mengisi Bahan Bakar";
        }
    }
}
```

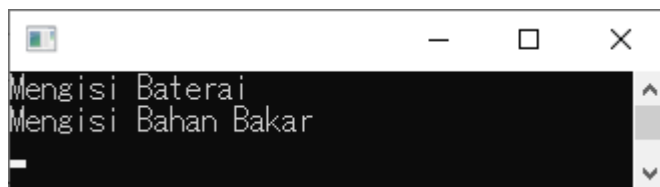
```

    }
}

class Program
{
    static void Main(string[] args)
    {
        MobilHybrid toyotaPrius = new MobilHybrid();
        Console.WriteLine(toyotaPrius.ChargeBaterai());
        Console.WriteLine(toyotaPrius.IsiBahanBakar());
        Console.ReadLine();
    }
}
}

```

Kode program di atas menghasilkan keluaran seperti yang ditampilkan pada gambar 9.3.



Gambar 9.3

Bab 10

Studi Kasus: Membuat Simulator Kendaraan

Bab ini saya tulis dengan gaya tutorial langkah demi langkah, agar Anda dapat langsung mempraktekan apa yang Anda baca di bab ini dengan menggunakan *Visual Studio*.

Saya menggunakan *Visual Studio Community 2019*. Apabila laptop/PC Anda belum terinstall *Visual Studio*, Anda bisa mengunduhnya melalui tautan di bawah ini secara gratis.

<https://visualstudio.microsoft.com/vs/>

Apakah Anda harus menggunakan versi *Visual Studio* yang sama dengan saya? Tidak.

Anda bisa menggunakan versi *Visual Studio* apapun. Namun, Anda akan melihat tampilan yang sedikit berbeda dengan *Visual Studio Community 2019*.

Mengidentifikasi sistem

Pada bab ini, kita akan mencoba membuat sebuah aplikasi simulator kendaraan sederhana. Tentunya, kita akan membuat sebuah aplikasi yang

tidak hanya *ekstensible* (dapat diperluas fungsinya), namun juga fleksibel.

Pada aplikasi ini, kita akan melibatkan beberapa objek seperti pengemudi dan kendaraan. Objek pengemudi ini nantinya juga harus mampu mengemudikan **semua** jenis kendaraan, baik kendaraan bermotor maupun tidak bermotor, kendaraan darat maupun kendaraan terbang.

Selain itu, untuk membuat sebuah kendaraan bermotor, baik itu kendaraan darat maupun kendaraan terbang, kita membutuhkan sebuah mesin.

Selanjutnya, kita akan membuat beberapa *class* dengan mengacu pada beberapa kebutuhan dan tuntutan di atas.

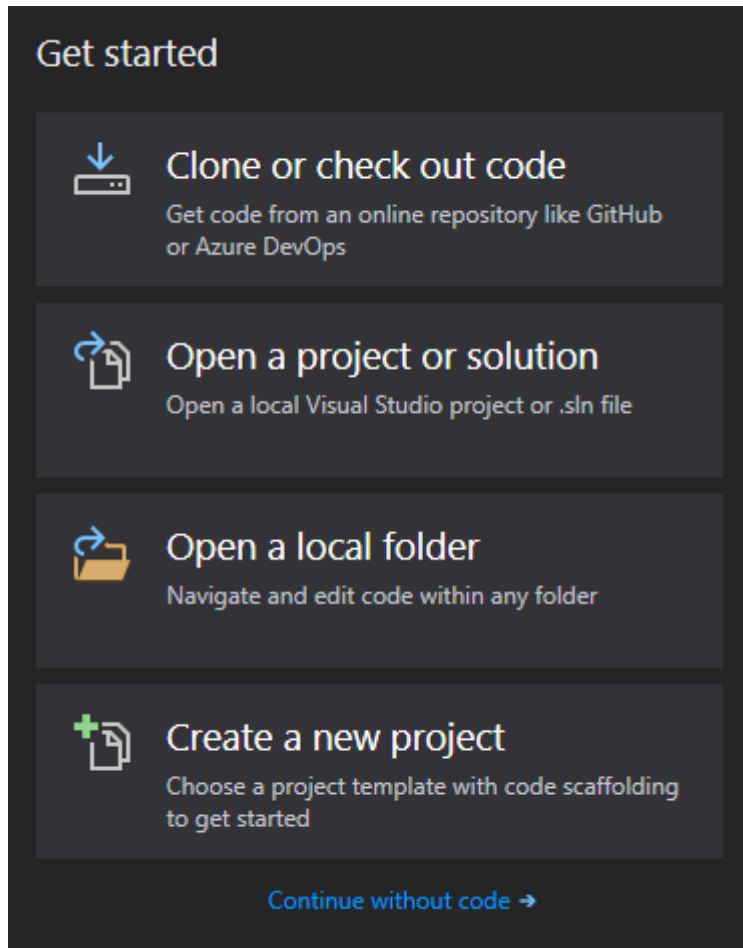
Mempersiapkan aplikasi di Visual Studio

Membuat aplikasi konsol

Aplikasi simulator kendaraan yang akan kita kembangkan bersama-sama di bab ini adalah berupa aplikasi konsol untuk sistem operasi Windows.

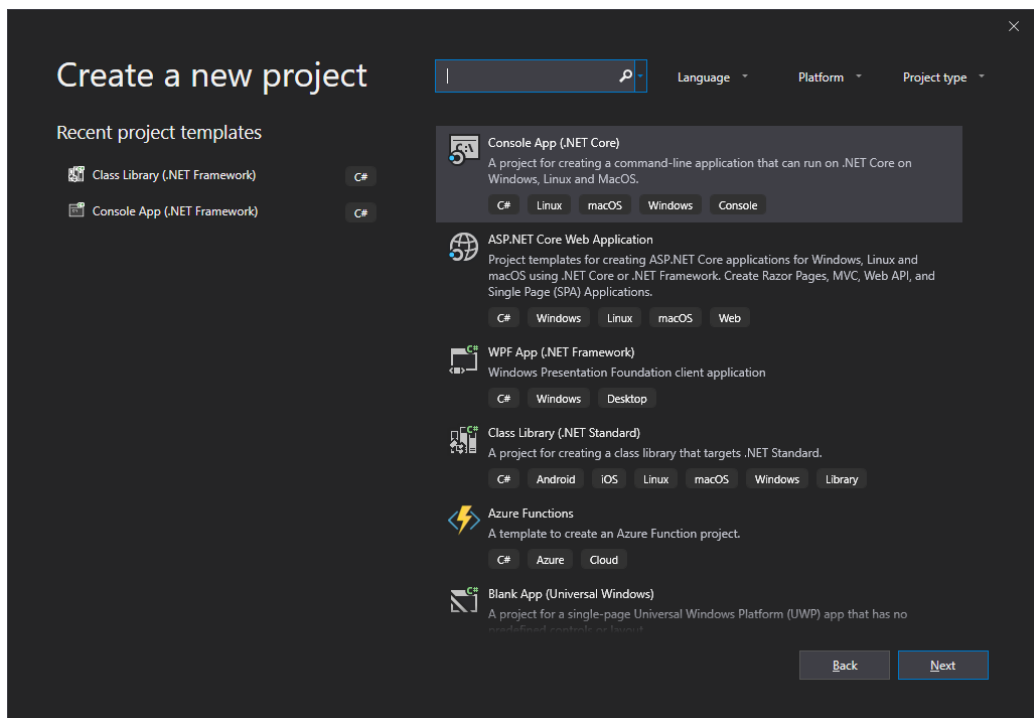
Ikuti langkah-langkah di bawah ini untuk membuat sebuah aplikasi konsol untuk sistem operasi Windows.

- 1) Buka Visual Studio Community 2019. Setelah terbuka, Anda akan menemui jendela *home screen*. Pada *home screen*, klik “*Create a new project*”.



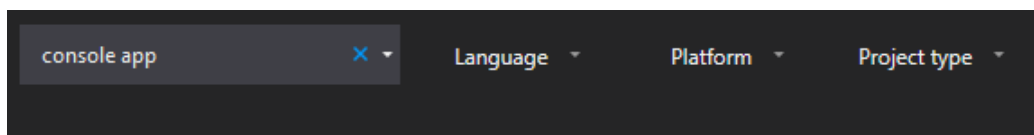
Gambar 10.1

- 2) Anda akan menemui tampilan seperti pada Gambar 10.2.



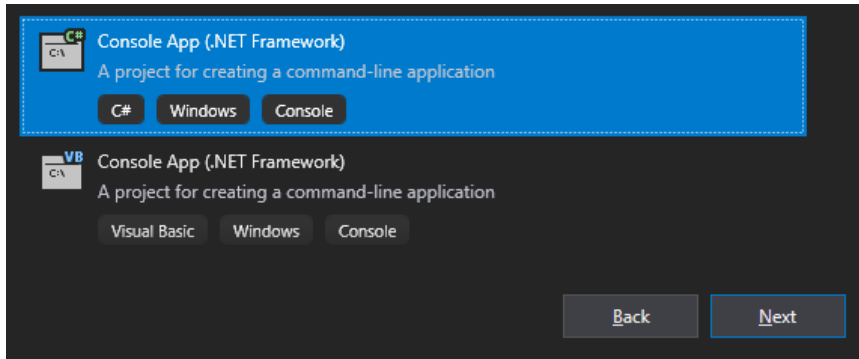
Gambar 10.2

- 3) Ketik “*console app*” di dalam kolom pencarian yang terletak di samping tulisan “*Create a new project*”.



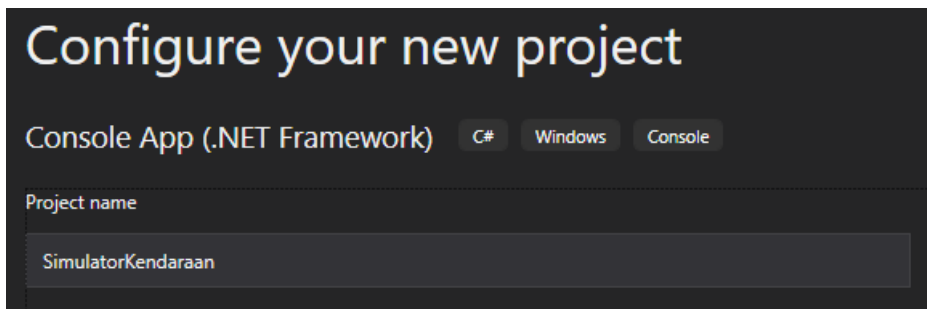
Gambar 10.3

- 4) Pilih **Console App (.NET Framework)**. Lalu, tekan tombol **Next**.



Gambar 10.4

- 5) Pada jendela “*Configure your new project*”, ketik “SimulatorKendaraan” di dalam kolom *Project Name*. Secara otomatis, kolom *Solution Name* akan terisi dengan nama yang sama.



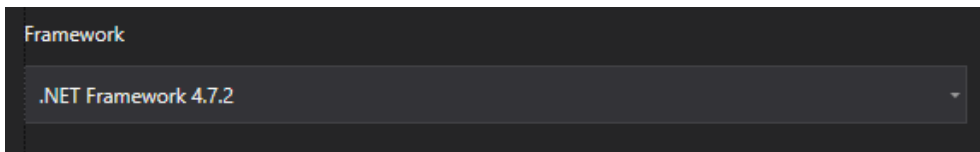
Gambar 10.4

- 6) Pilih lokasi di mana Anda ingin menyimpan *project* Anda.



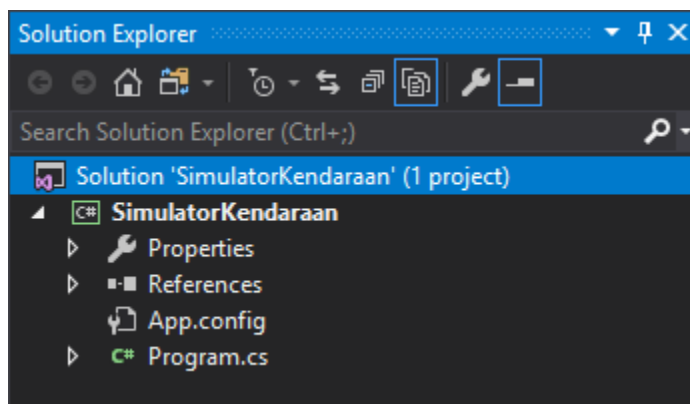
Gambar 10.5

- 7) Untuk sementara, Anda bisa mengabaikan kolom *Framework*. Saya sendiri menggunakan .NET Framework 4.7.2 yang merupakan versi .NET terbaru yang didukung oleh Visual Studio 2019.



Gambar 10.6

- 8) Tekan tombol **Create** untuk membuat aplikasi konsol Anda.
- 9) Setelah aplikasi konsol Anda berhasil dibuat, Anda akan mendapatkan sebuah file bernama *Program.cs*.



Gambar 10.7

Membuat class library

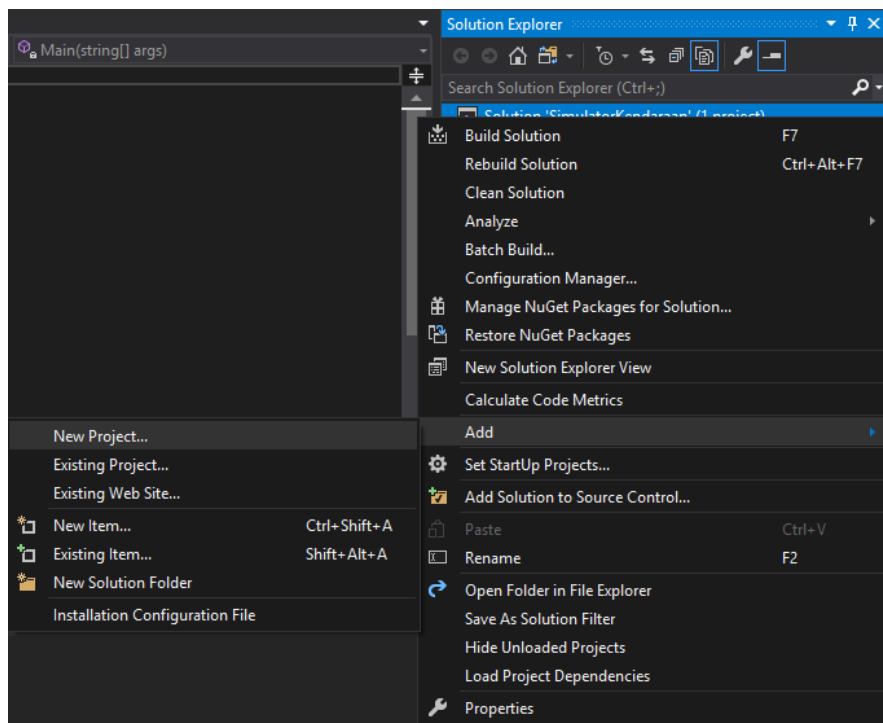
Kita akan memisahkan semua *class* yang kita butuhkan dari aplikasi konsol yang barusan kita buat. Untuk itu, kita membutuhkan sebuah pustaka yang

disebut dengan *class library*.

Apabila aplikasi konsol yang sebelumnya Anda buat nantinya akan dikompilasi sebagai *executable file* atau file *.exe*, maka *class library* ini nantinya akan dikompilasi sebagai *dynamic link library* atau file *.dll*.

Ikuti langkah-langkah di bawah ini untuk membuat sebuah *class library*.

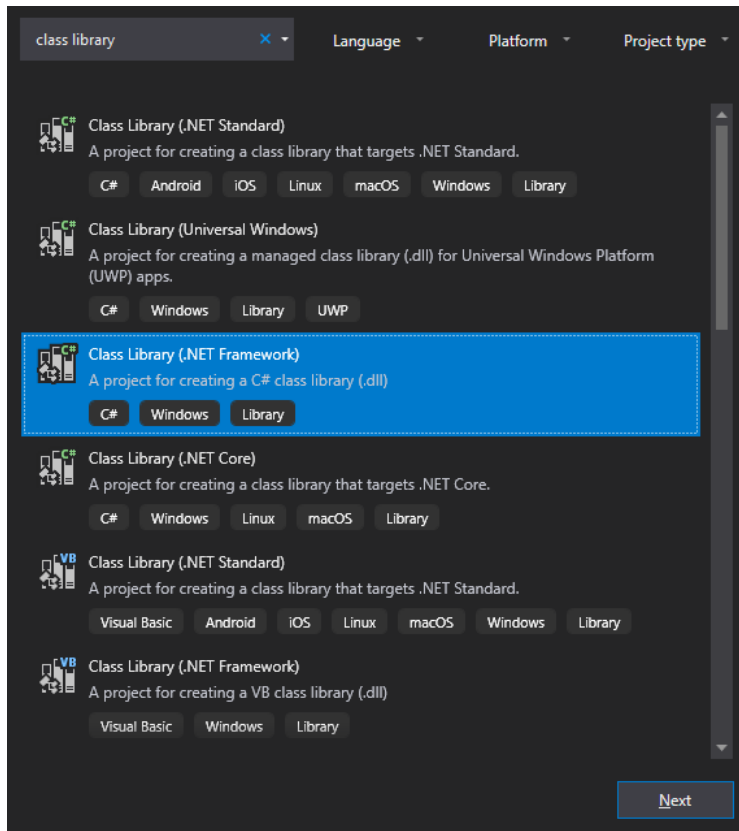
- 1) Arahkan kursor Anda pada *Solution Explorer*. Klik kanan pada ***Solution ‘SimulatorKendaraan’***.



Gambar 10.8

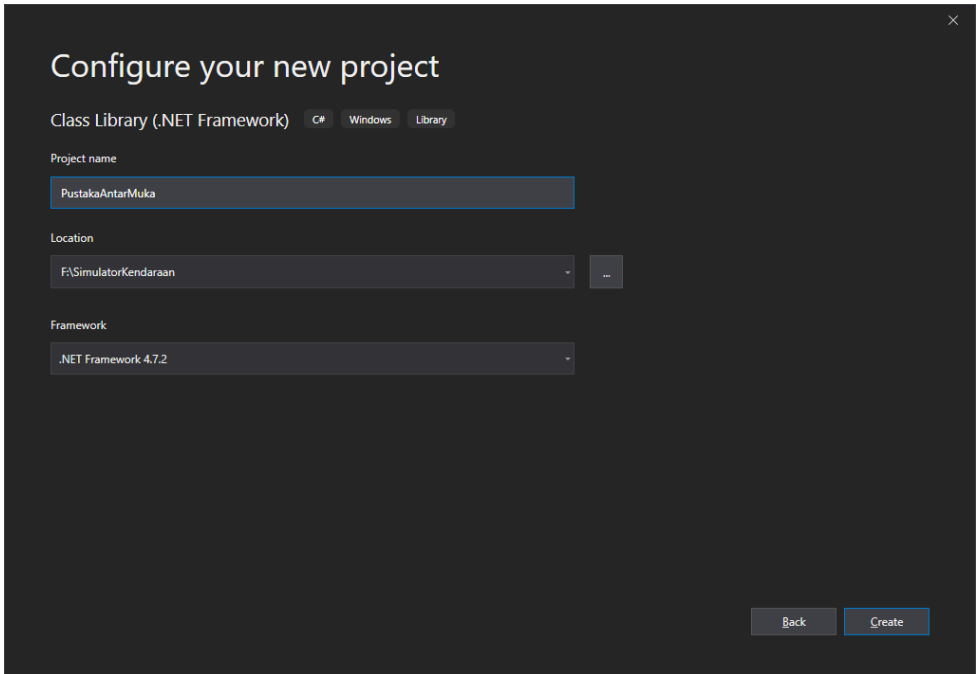
- 2) Pada *pop-up* yang muncul, tekan **add -> new project**.

- 3) Setelah jendela “*Create a new project*” muncul, ketik “*class library*” pada kolom pencarian. Pilih ***Class Library (.NET Framework)***. Lalu, tekan tombol ***Next***.



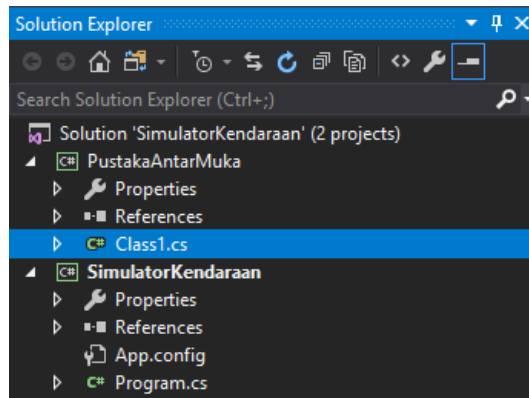
Gambar 10.9

- 4) Setelah Anda menemui tampilan seperti pada Gambar 10.10, ketik “PustakaAntarMuka” pada kolom *Project Name*. Lalu, tekan tombol **Create**.



Gambar 10.10

- 5) Setelah *class library* Anda berhasil dibuat, Anda akan mendapatkan sebuah file bernama *Class1.cs* di dalam pustaka bernama *PustakaAntarMuka*.



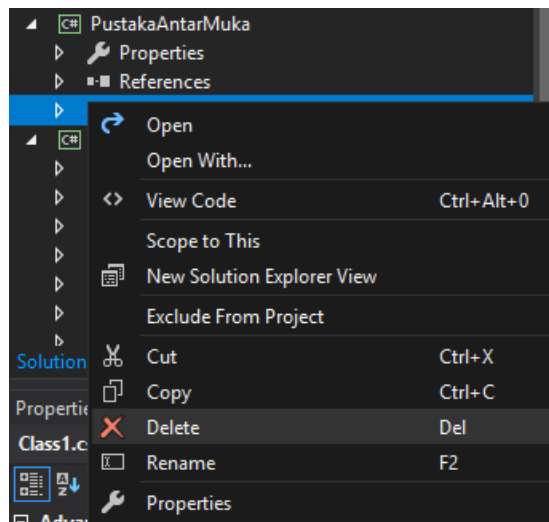
Gambar 10.11

- 6) Ulangi langkah 1-5 untuk membuat dua lagi *class library*. Kali ini, beri nama masing-masing dengan “PustakaAktor” dan “PustakaKendaraan”.

Menambahkan interface ke dalam class library

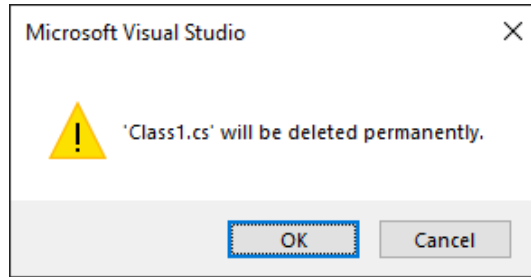
Aplikasi simulator kendaraan yang akan kita kembangkan, membutuhkan satu atau lebih *interface*. Oleh karena itu, Kita perlu menambahkan *interface* ke dalam *class library* PustakaAntarMuka yang sebelumnya telah Anda buat.

- 1) Sebelum menambahkan file *interface*, kita akan menghapus file *Class1.cs* yang tidak kita butuhkan di dalam PustakaAntarMuka terlebih dahulu. Caranya, klik kanan pada *Class1.cs*, lalu tekan **Delete**.



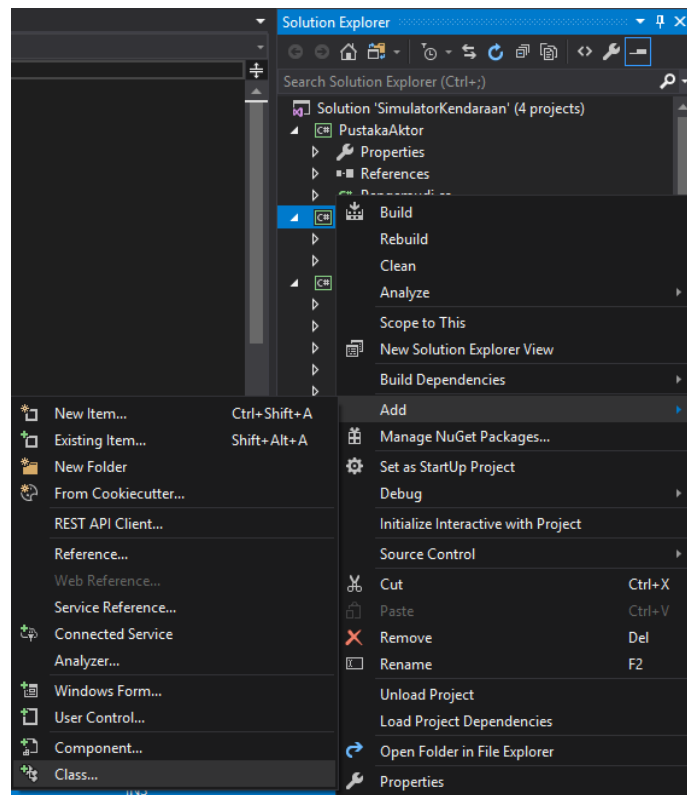
Gambar 10.12

- 2) Anda akan diberi peringatan bahwa file *Class1* akan dihapus secara permanen. Tekan tombol **OK**.



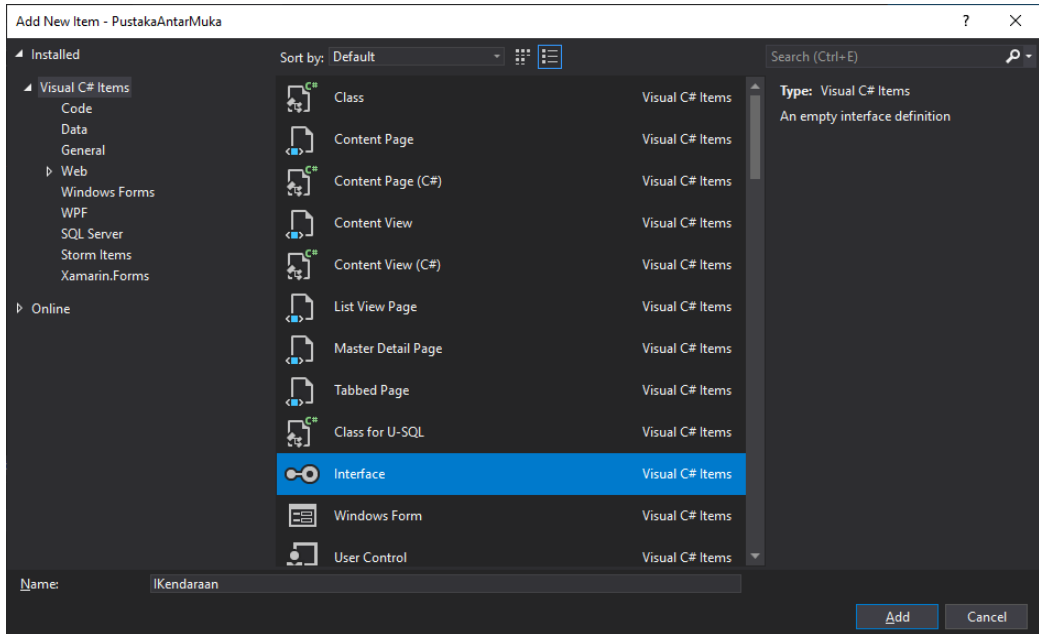
Gambar 10.13

- 3) Untuk menambahkan sebuah *interface*, klik kanan pada *Project* *PustakaAntarMuka* untuk menampilkan *pop-up*. Pada *pop-up*, tekan **add -> class**.



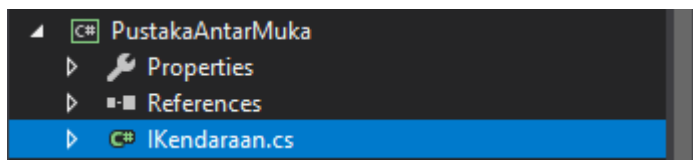
Gambar 10.14

- 4) Pada jendela *Add New Item*, pilih *interface*. Ketik “IKendaraan” pada kolom *Name*, lalu tekan tombol **Add**.



Gambar 10.15

- 5) Anda akan mendapatkan file *IKendaraan.cs* di dalam *class library* PustakaAntarMuka.



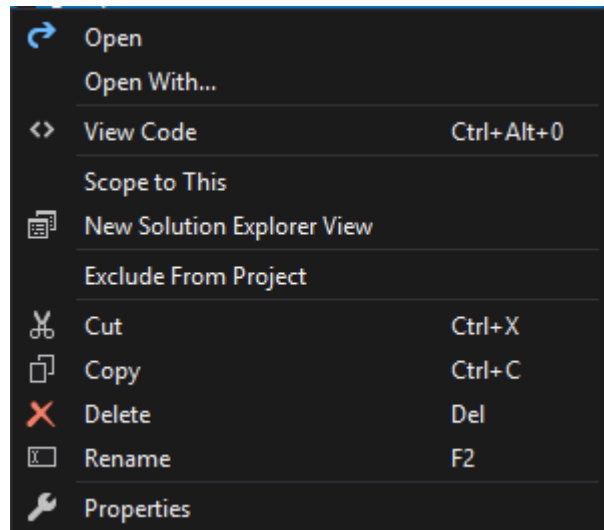
Gambar 10.16

Merubah nama class

Setelah Anda membuat sebuah *class library*, Visual Studio secara default

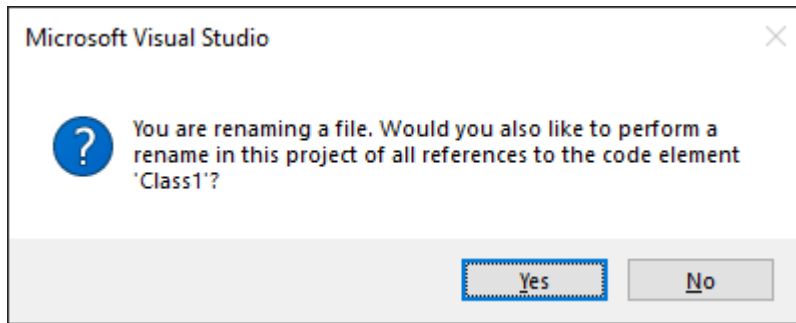
akan memberi Anda sebuah file bernama *Class1.cs*. Anda bisa merubah nama *Class1* ke nama yang lebih masuk akal dengan cara seperti berikut ini.

- 1) Klik kanan pada file *Class1.cs*.
- 2) Pada pop-up, tekan **Rename**. Tulis nama file yang Anda inginkan. Lalu tekan Enter.



Gambar 10.17

- 3) Visual Studio akan mengkonfirmasi Anda, apakah Anda juga ingin merubah nama semua referensi yang mengacu pada *Class1*. Tekan tombol **Yes**.



Gambar 10.18

Dengan begitu, Anda tidak hanya merubah nama filenya saja. Namun, Anda juga merubah nama *class*-nya.

Gunakan langkah-langkah di atas untuk merubah nama file *Class1.cs* pada *PustakaAktor* menjadi *Pengemudi.cs*.

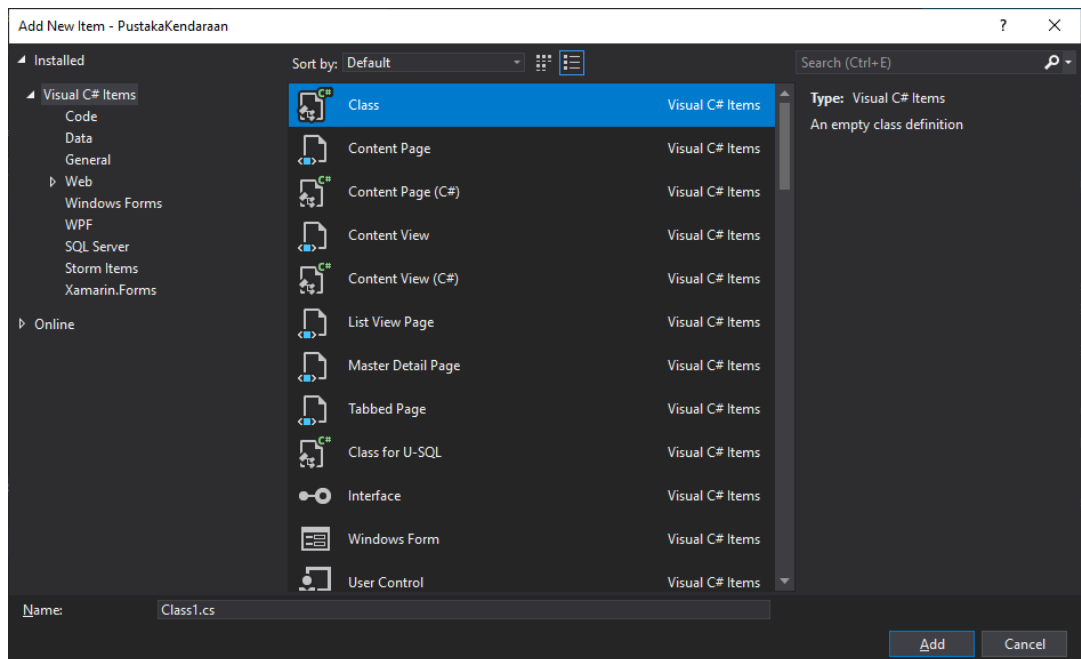
Menambahkan class ke dalam class library

Class library merupakan sebuah pustaka yang digunakan untuk mengumpulkan beberapa *class* yang berhubungan. Sejauh ini, aplikasi kita memiliki tiga buah *class library*, yaitu **PustakaAntarMuka**, **PustakaAktor**, dan **PustakaKendaraan**.

Class library **PustakaAntarMuka** kita gunakan untuk mengumpulkan beberapa *interface* yang dibutuhkan di dalam aplikasi kita. Sementara, *class library* **PustakaAktor** kita gunakan untuk mengumpulkan beberapa *class* yang berhubungan dengan aktor-aktor di dalam aplikasi kita, seperti pengemudi. Terakhir, *class library* **PustakaKendaraan** kita gunakan untuk mengumpulkan beberapa *class* yang berhubungan dengan jenis-jenis

kendaraan yang ada di dalam aplikasi kita nantinya.

Untuk menambahkan sebuah *class* ke dalam pustaka, Anda bisa melakukannya seperti pada saat menambahkan *interface* ke dalam PustakaAntarMuka. Hanya saja di jendela *Add New Item*, jika sebelumnya Anda memilih *interface*, kali ini pilih *class*.



Gambar 10.19

Sejauh ini, Anda sudah membuat sebuah *interface* *IKendaraan* di dalam PustakaAntarMuka dan juga sebuah *class* *Pengemudi* di dalam PustakaAktor. Selanjutnya, Anda perlu menambahkan *class-class* di bawah ini ke dalam *class library* PustakaKendaraan.

a) Mobil

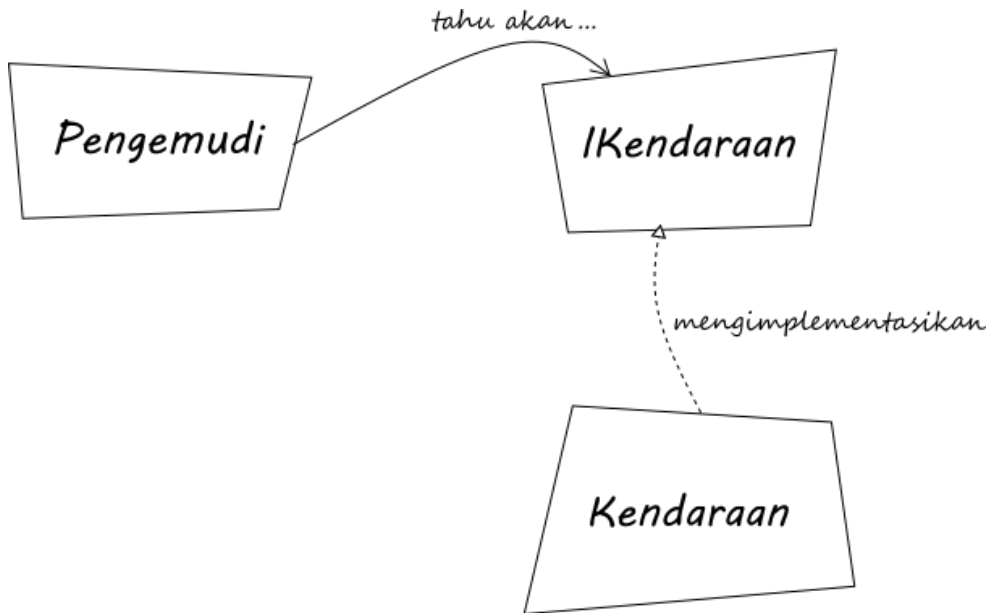
- b) Motor
- c) Pesawat
- d) Becak
- e) Mesin

Antarmuka untuk interaksi antar objek

Di dalam aplikasi simulator kendaraan ini, kita akan membuat **Pengemudi** bisa mengoperasikan seluruh jenis kendaraan, tidak peduli itu adalah kendaraan bermotor ataupun bukan, kendaraan darat ataupun kendaraan terbang.

Dengan tuntutan seperti itu, berarti **Pengemudi** perlu berinteraksi dengan sebuah antarmuka yang umum untuk semua jenis kendaraan. Sehingga, selama jenis kendaraan yang dia kemudikan mengimplementasikan antarmuka tersebut, **Pengemudi** tidak akan gagal mengemudikannya.

Jika Anda perhatikan diagram pada gambar 10.20, **Pengemudi** mengerti bagaimana berinteraksi dengan sebuah antarmuka bernama **IKendaraan**. Sehingga setiap jenis kendaraan yang mengimplementasikan antarmuka **IKendaraan**, dapat dikemudikan oleh **Pengemudi**.



Gambar 10.20

Kembali ke Visual Studio, buka file *IKendaraan.cs* yang sudah Anda buat sebelumnya. Tambahkan **public** sebagai *access modifier* di depan kata kunci **interface**. Lalu tambahkan kode program di bawah ini ke dalam badan *interface*.

```
void Jalankan();
```

Kode program di atas merupakan kontrak yang ditentukan oleh **IKendaraan**. Yaitu, setiap *class* yang menggunakan **IKendaraan** sebagai *interface*, harus mengimplementasikan *method* `Jalankan()`.

Dengan demikian kode program untuk **IKendaraan** akan menjadi seperti di bawah ini.

```
namespace PustakaAntarMuka
{
    public interface IKendaraan
    {
        void Jalankan();
    }
}
```

Mengidentifikasi relasi antar class

Sampai saat ini, Anda tahu bahwa aplikasi Anda membutuhkan beberapa *class*. Tentu saja *class-class* tersebut memiliki relasi antara satu dengan lainnya.

Misalnya, apa hubungan antara *class Mesin* dengan *class Mobil*? Apakah objek dari *class Mesin* dan *class Mobil* adalah dua objek yang saling tergantung atau tidak? Lalu, apakah objek mesin tersebut bisa digunakan oleh mobil dan sekaligus oleh pesawat?

Pewarisan

Menurut Anda, apakah objek mesin bisa digunakan oleh kendaraan darat dan kendaraan udara? *class Mesin* untuk kendaraan darat dan udara seharusnya memiliki mekanisme yang berbeda, bukan?

Oleh karena itu, kita perlu membuat cetak biru yang berbeda untuk mesin kendaraan darat dan mesin kendaraan udara. Namun demikian, keduanya tetap merupakan sebuah mesin yang bisa dihidupkan, dimatikan, dan juga dipicu untuk berakselerasi.

Seperti yang sudah Anda pelajari, hubungan seperti di atas adalah hubungan antara entitas induk dengan entitas anak, atau pewarisan.

Untuk menerapkannya di dalam aplikasi kita, tambahkan dua *class* di bawah ini ke dalam *class library* PustakaKendaraan.

a) MesinDarat

b) MesinTerbang

Setelah Anda menambahkan kedua *class* di atas, selanjutnya kita akan menulis kode program untuk *class* **Mesin**, *class* **MesinDarat**, dan *class* **MesinTerbang**. *Class* **Mesin** adalah sebuah *abstract class*, sedangkan *class* **MesinDarat** dan **MesinTerbang** merupakan turunan dari *class* **Mesin**.

Mesin.cs

```
using System;

namespace PustakaKendaraan
{
    public abstract class Mesin
    {
        public abstract void Hidupkan();

        public abstract void Matikan();

        public void Picu(string pemicu)
        {
            Console.WriteLine(pemicu);
        }
    }
}
```

MesinDarat.cs

```
using System;

namespace PustakaKendaraan
{
    public class MesinDarat : Mesin
    {
        public override void Hidupkan()
        {
            Console.WriteLine("Menghidupkan mesin darat");
        }

        public override void Matikan()
        {
            Console.WriteLine("Mematikan mesin darat");
        }
    }
}
```

MesinTerbang.cs

```
using System;

namespace PustakaKendaraan
{
    public class MesinTerbang : Mesin
    {
        public override void Hidupkan()
        {
            Console.WriteLine("Menghidupkan mesin terbang");
        }

        public override void Matikan()
        {
            Console.WriteLine("Mematikan mesin terbang");
        }
    }
}
```

```
}
}
```

Asosiasi

Di dalam aplikasi kita, pengemudi dan antarmuka kendaraan merupakan dua entitas yang independen. Artinya, eksistensi dari antarmuka kendaraan tidak terpengaruh oleh eksistensi pengemudi.

Katakanlah ketika pengemudi keluar dari mobil (mengakhiri sesi mengemudi), apakah kemudian stir kemudi, pedal gas, dan tuas transmisi juga akan hilang? Tentunya tidak, bukan?

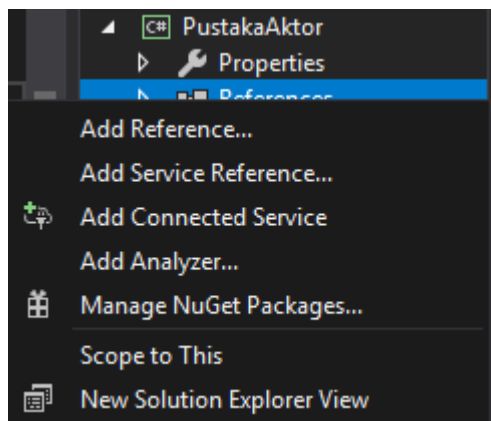
Selain itu, seorang pengemudi bisa mengendarai mobil manapun. Begitu pula, sebuah mobil juga bisa dikendarai oleh pengemudi siapapun.

Relasi yang diperlukan antara pengemudi dan antarmuka kendaraan hanyalah membuat pengemudi mengerti bagaimana berinteraksi dengan antarmuka kendaraan agar dapat mengoperasikannya. Relasi seperti ini biasa disebut dengan asosiasi.

Sekarang, kita akan menerapkan asosiasi antara *class* `Pengemudi` dengan *interface* `IKendaraan`. Namun, sebelum Anda menulis kode program untuk *class* `Pengemudi`, Anda diharuskan untuk menambahkan referensi ke *interface* `IKendaraan` pada *class library* `PustakaAktor` di mana *class* `Pengemudi` berada.

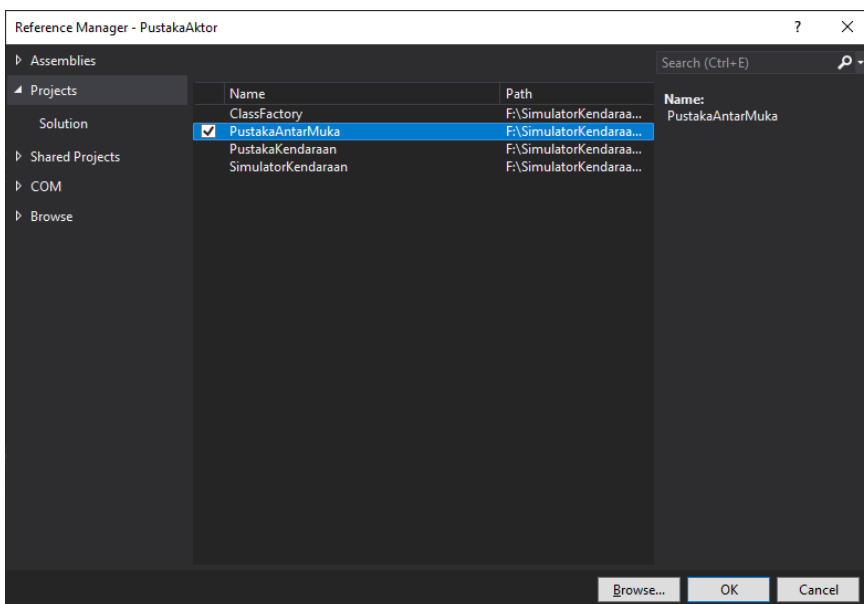
1. Pada *class library* `PustakaAktor`, klik kanan pada *reference*, lalu pilih

add reference.



Gambar 10.21

2. Pada jendela *Reference Manager*, centang *PustakaAntarMuka*, lalu tekan tombol OK.



Gambar 10.22

3. *Class library* `PustakaAntarMuka` akan ditambahkan ke dalam referensi di dalam *class library* `PustakaKendaraan`. Dengan demikian, Anda sekarang bisa menggunakan `PustakaAntarMuka` pada *class-class* di dalam `PustakaKendaraan`.
4. Agar Anda bisa mengakses elemen-elemen di dalam `PustakaAntarMuka` yang dideklarasikan sebagai **public**, tambahkan kode program di bawah ini di bagian sebelum kata kunci `namespace`.

```
using PustakaAntarMuka;
```

Kode program lengkap untuk *class* `Pengemudi` menjadi seperti berikut ini.

Pengemudi.cs

```
using PustakaAntarMuka;
```

```
namespace PustakaAktor
{
    public class Pengemudi
    {
        public void Mengemudi (IKendaraan _kendaraan)
        {
            _kendaraan. Jalankan();
        }
    }
}
```

Apabila Anda perhatikan, tidak ada objek dari tipe `IKendaraan` yang diinstansiasi dari *class* `Pengemudi`. `IKendaraan` hanya kita berikan kepada *class* `Pengemudi` sebagai parameter dari *method* `Mengemudi()`.

Artinya, objek dari *class* `Pengemudi` hanya mengerti bahwa ketika dia mengeksekusi *method* `Mengemudi()`, maka objek dari tipe `IKendaraan` harus diberikan sebagai parameter. Dia tidak bertanggung jawab untuk menginstansiasi objek dari tipe `IKendaraan`.

Komposisi

Anda tentunya sudah tahu bahwa kendaraan bermotor adalah kendaraan yang digerakkan oleh tenaga mesin, sementara kendaraan tidak bermotor digerakkan oleh tenaga manusia.

Dengan kata lain, mesin adalah salah satu komponen penyusun (bagian dari) sebuah kendaraan bermotor. Objek kendaraan bermotor dan objek mesin adalah dua objek yang saling bergantung. Ketika sebuah kendaraan bermotor dikonstruksi, maka sebuah mesin juga harus dikonstruksi. Hubungan seperti ini biasa disebut dengan komposisi.

Di dalam aplikasi ini, kita memiliki tiga jenis kendaraan bermotor, yaitu mobil, motor, dan pesawat. Mobil dan motor adalah kendaraan bermotor yang hanya bisa berjalan di darat, sedangkan pesawat adalah tipe kendaraan bermotor yang bisa diterbangkan.

Sebelumnya, Anda telah membuat *class* `MesinDarat` dan *class* `MesinTerbang`. Kita akan mengkomposisikan `MesinDarat` ke `Mobil` dan `Motor`. Sementara `MesinTerbang` akan kita komposisikan ke `Pesawat`.

Mobil.cs

```

using System;

namespace PustakaKendaraan
{
    public class Mobil
    {
        private readonly MesinDarat _mesinMobil;

        public Mobil()
        {
            _mesinMobil = new MesinDarat();
        }
    }
}

```

Motor.cs

```

using System;

namespace PustakaKendaraan
{
    public class Motor
    {
        private readonly MesinDarat _mesinMotor;

        public Motor()
        {
            _mesinMotor = new MesinDarat();
        }
    }
}

```

Pesawat.cs

```

using System;

namespace PustakaKendaraan

```

```

{
    public class Pesawat
    {
        private readonly MesinTerbang _mesinPesawat;

        public Pesawat()
        {
            _mesinPesawat = new MesinTerbang();
        }
    }
}

```

Perhatikan bahwa kali ini, objek dari tipe `MesinDarat` dan `MesinTerbang` diinstansiasi di dalam *constructor*. Konsekuensinya adalah ketika sebuah objek, misalnya, dari tipe `Mobil` diinstansiasi, maka objek `MesinDarat` juga ikut diinstansiasi. Eksistensi objek dari *class* `MesinDarat` sangat bergantung pada eksistensi objek dari *class* `Mobil`.

Mengimplementasikan interface

Perlu diingat bahwa pengemudi hanya mengerti cara berinteraksi dengan antarmuka kendaraan untuk mengemudikan sebuah kendaraan. Oleh karena itu, agar pengemudi bisa mengemudikan setiap kendaraan yang kita punya, maka kita harus mengimplementasikan *interface* `IKendaraan` untuk setiap kendaraan yang kita punya.

Mobil.cs

```

using System;
using PustakaAntarMuka;

```

```

namespace PustakaKendaraan
{
    public class Mobil : IKendaraan
    {
        private readonly MesinDarat _mesinMobil;

        public Mobil()
        {
            _mesinMobil = new MesinDarat();
        }

        public void Jalankan()
        {
            Console.WriteLine($"Menjalankan Mobil");

            _mesinMobil.Hidupkan();

            _mesinMobil.Picu("Injak pedal gas");

            _mesinMobil.Matikan();
        }
    }
}

```

Motor.cs

```

using System;
using PustakaAntarMuka;

namespace PustakaKendaraan
{
    public class Motor : IKendaraan
    {
        private readonly MesinDarat _mesinMotor;

        public Motor()
        {

```

```
        _mesinMotor = new MesinDarat();
    }

    public void Jalankan()
    {
        Console.WriteLine("Menjalankan Motor");

        _mesinMotor.Hidupkan();

        _mesinMotor.Picu("Putar handle gas");

        _mesinMotor.Matikan();
    }
}
}
```

Pesawat.cs

```
using System;
using PustakaAntarMuka;

namespace PustakaKendaraan
{
    public class Pesawat : IKendaraan
    {
        private readonly MesinTerbang _mesinPesawat;

        public Pesawat()
        {
            _mesinPesawat = new MesinTerbang();
        }

        public void Jalankan()
        {
            Console.WriteLine("Menjalankan Pesawat");

            _mesinPesawat.Hidupkan();
        }
    }
}
```

```

        _mesinPesawat.Picu("Tarik handle");

        Console.WriteLine("take off");
        Console.WriteLine("terbang di angkasa");
        Console.WriteLine("mendarat dengan selamat");

        _mesinPesawat.Matikan();
    }
}

```

Becak.cs

```

using System;
using PustakaAntarMuka;

namespace PustakaKendaraan
{
    public class Becak : IKendaraan
    {
        public void Jalankan()
        {
            Console.WriteLine("¥nMenjalankan Becak");
            Console.WriteLine("Mengayuh Becak keliling kota");
        }
    }
}

```

Menyembunyikan class dari dunia luar

Anda tahu bahwa sebuah mesin adalah komponen internal dari sebuah kendaraan. Disamping itu, mesin adalah komponen yang cukup rumit. Membiarkan mesin bisa diakses dan digunakan oleh siapa saja, bisa saja

membahayakan seseorang yang mengemudikannya. Sabotase, misalnya.

Oleh karena itu, kita ingin menyembunyikan menyembunyikan mesin dari dunia luar. Membuatnya tidak bisa diakses kecuali dari dalam *assembly*-nya.

Sebelumnya, kita telah mendeklarasikan *abstract class* `Mesin` dan juga turunannya, yaitu *class* `MesinDarat` dan *class* `MesinTerbang` sebagai **public**. Artinya, kita mengekspos *class-class* tersebut ke dunia luar.

Untuk membuat *class-class* tersebut hanya bisa diakses dari dalam *assembly*-nya saja, ganti *access modifier* **public** yang sebelumnya kita tetapkan menjadi *access modifier* **internal** seperti kode program di bawah ini.

Mesin.cs

```
internal abstract class Mesin
{
    internal abstract void Hidupkan();

    internal abstract void Matikan();

    internal void Picu(string pemicu)
    {
        Console.WriteLine(pemicu);
    }
}
```

MesinDarat.cs

```
internal class MesinDarat : Mesin
{
    internal override void Hidupkan()
```



```

{
    Console.WriteLine("Menghidupkan mesin darat");
}

internal override void Matikan()
{
    Console.WriteLine("Mematikan mesin darat");
}
}

```

MesinTerbang.cs

```

internal class MesinTerbang : Mesin
{
    internal override void Hidupkan()
    {
        Console.WriteLine("Menghidupkan mesin terbang");
    }

    internal override void Matikan()
    {
        Console.WriteLine("Mematikan mesin terbang");
    }
}

```

Dengan demikian, ketiga *class* di atas sekarang hanya bisa diakses dari dalam *assembly*-nya saja, yaitu kode program yang masih berada di dalam *namespace* *PustakaKendaraan*. Kode program di luar *namespace* *PustakaKendaraan*, tidak lagi bisa mengakses ketiga *class* tersebut secara langsung.

Mengkapsulasi proses instansiasi objek

Pada aplikasi konsol yang akan kita buat nanti, kita menginginkan aplikasi

konsol yang fleksibel. Artinya, kita tidak ingin menulis dan mengkompilasi ulang kode program pada aplikasi konsol tiap kali kita menambahkan tipe kendaraan baru pada *class library* `PustakaKendaraan`.

Sekali aplikasi konsol kita dikompilasi dan dirilis, aplikasi tersebut harus bisa menyesuaikan perubahan yang dilakukan pada `PustakaKendaraan`.

Umumnya di dalam *method* `Main()` pada *class* `Program`, kita akan menulis kode program seperti berikut ini.

```
class Program
{
    static void Main(string[] args)
    {
        IKendaraan mobil = new Mobil();
        IKendaraan motor = new Motor();

        Pengemudi pengemudi = new Pengemudi();

        pengemudi.Mengemudi(mobil);
        pengemudi.Mengemudi(motor);
    }
}
```

Katakanlah pada *class library* `PustakaKendaraan`, kita baru memiliki dua buah tipe kendaraan, yaitu `Mobil` dan `Motor`. Lalu, ketika kita ingin mengembangkan aplikasi kita dengan menambahkan `Pesawat` dan `Becak` ke dalam `PustakaKendaraan`, sudah pasti kita juga harus menulis dan mengkompilasi ulang aplikasi konsol kita.

Untuk mengatasi hal ini, kita akan memilih untuk mengenkapsulasi proses

instansiasi objek ke dalam sebuah *class* terpisah. *Class* ini nantinya bertanggung jawab untuk menginstansiasi objek dari tipe kendaraan yang diinginkan.

Karena *class* ini hanya berfungsi untuk menginstansiasi sebuah objek, maka *class* ini biasa disebut dengan *Class Factory*. Untuk kasus kita kali ini, kita akan menggunakan pola *simple factory method*.

Catatan: *Simple factory method* bukan merupakan salah satu pola desain (*design patterns*) yang ditulis *Gang of Four* di dalam buku *Design Patterns: Elements of Reusable Object-Oriented Software*. Pola-pola desain pemrograman berorientasi objek sendiri di luar cakupan dari buku ini. Materi ini nantinya akan saya sampaikan di buku atau kursus tersendiri.

Sebelum memulai menulis kode program, buat sebuah *class library* baru bernama `ClassFactory`, kemudian buat sebuah *class* bernama `FactoryKendaraan` di dalam pustaka tersebut. Anda bisa mengikuti langkah-langkah yang telah dijelaskan sebelumnya.

Setelah itu Anda perlu menambahkan referensi ke `PustakaAntarMuka` dan `PustakaKendaraan` ke dalam *class library* `ClassFactory`.

Buka file `FactoryKendaraan.cs`, tambahkan `using PustakaAntarMuka;` dan

`using` `PustakaKendaraan`; di bagian header, lalu tulis kode program di bawah ini di dalam badan `class` `FactoryKendaraan`.

```
public IKendaraan Create(int pilihan)
{
    IKendaraan kendaraan;

    switch (pilihan)
    {
        case 1:
            kendaraan = new Mobil();
            break;
        case 2:
            kendaraan = new Motor();
            break;
        case 3:
            kendaraan = new Pesawat();
            break;
        case 4:
            kendaraan = new Becak();
            break;
        default:
            throw new ArgumentException();
    }

    return kendaraan;
}
```

Perhatikan kembali kode program di atas. *Method* `Create()` mengambil satu buah parameter bertipe data `integer`. Nantinya, argumen yang diberikan melalui parameter tersebut digunakan untuk menentukan tipe kendaraan apa yang perlu dibuat. Selanjutnya, *method* tersebut akan mengembalikan sebuah nilai dengan tipe `IKendaraan`.

Mengonsumsi class library

Kini saatnya kita menulis kode program untuk aplikasi konsol yang sudah kita buat.

Agar kita bisa mengakses *class-class* dan *interface* yang sudah kita buat sebelumnya, kita perlu menambahkan `PustakaAntarMuka`, `PustakaAktor`, dan `ClassFactory` sebagai referensi untuk aplikasi kita (*project* `SimulatorKendaraan`).

Sebenarnya, kita juga bisa menambahkan `PustakaKendaraan` sebagai referensi. Namun, kita ingin aplikasi kita tidak bergantung langsung dengan pustaka tersebut. Sebagai gantinya, kita menggunakan `ClassFactory` untuk mengakses *class-class* di dalam `PustakaKendaraan`.

Program.cs

```
using System;
using ClassFactory;
using PustakaAktor;
using PustakaAntarMuka;

namespace SimulatorKendaraan
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("AKSI PENGEMUDI");
            Console.Write("Pilih jenis kendaraan : ");

            int pilihanKendaraan = Convert.ToInt32(Console.ReadLine());
```

```
FactoryKendaraan objKendaraan = new FactoryKendaraan();

IKendaraan kendaraan = objKendaraan.Create(pilihanKendaraan);

Pengemudi pengemudi = new Pengemudi();

pengemudi.Mengemudi(kendaraan);

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

Pada kode program di atas, kita mengambil masukan bertipe `integer` yang diberikan oleh pengguna ke aplikasi kita.

```
int pilihanKendaraan = Convert.ToInt32(Console.ReadLine());
```

Lalu, kita menginstansiasi objek dari tipe `FactoryKendaraan`.

```
FactoryKendaraan objKendaraan = new FactoryKendaraan();
```

Selanjutnya, kita menetapkan nilai dari variabel kendaraan yang bertipe `IKendaraan` dengan nilai yang dikembalikan oleh pemanggilan *method* `Create()` dari objek `objKendaraan`.

```
IKendaraan kendaraan = objKendaraan.Create(pilihanKendaraan);
```

Baris berikutnya adalah instansiasi objek pengemudi dari tipe `Pengemudi`.

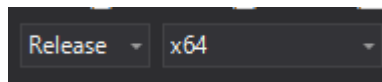
```
Pengemudi pengemudi = new Pengemudi();
```

Terakhir... kita buat pengemudi mengemudikan kendaraan dari tipe `IKendaraan`.

`pengemudi.Mengemudi(kendaraan)` ;















Mengkompilasi dan menjalankan program

- 1) Pilih *Release* pada *Solution Configuration* dan *x64* pada *target platform*.



Gambar 20.23

- 2) lalu tekan **ctrl+shift+B** (*build solution*).
- 3) Apabila tidak ada error pada saat proses *build*, Anda akan mendapatkan file-file seperti pada Gambar 10.24 di dalam folder `...\SimulatorKendaraan\bin\Release`.

名前	更新日時	種類	サイズ
 ClassFactory.dll	2019/08/19 15:44	アプリケーション拡張	5 KB
 ClassFactory.pdb	2019/08/19 15:44	PDB ファイル	14 KB
 PustakaAktor.dll	2019/08/19 15:44	アプリケーション拡張	4 KB
 PustakaAktor.pdb	2019/08/19 15:44	PDB ファイル	14 KB
 PustakaAntarMuka.dll	2019/08/19 15:44	アプリケーション拡張	4 KB
 PustakaAntarMuka.pdb	2019/08/19 15:44	PDB ファイル	8 KB
 PustakaKendaraan.dll	2019/08/19 15:44	アプリケーション拡張	6 KB
 PustakaKendaraan.pdb	2019/08/19 15:44	PDB ファイル	24 KB
 SimulatorKendaraan.exe	2019/08/19 15:44	アプリケーション	5 KB
 SimulatorKendaraan.exe.config	2019/08/07 21:11	XML Configuratio...	1 KB
 SimulatorKendaraan.pdb	2019/08/19 15:44	PDB ファイル	12 KB
 SimulatorKendaraan.vshost.exe	2019/08/19 15:44	アプリケーション	23 KB
 SimulatorKendaraan.vshost.exe.config	2019/08/07 21:11	XML Configuratio...	1 KB
 SimulatorKendaraan.vshost.exe.manifest	2018/09/15 16:29	MANIFEST ファイル	1 KB

Gambar 10.24

Jalankan *SimulatorKendaraan.exe*, lalu beri nilai masukan 1, 2, 3, atau 4. Anda akan melihat bahwa objek pengemudi adalah seorang *super driver*. Dia bisa mengendarai segala jenis kendaraan dari becak sampai pesawat. Ya, mirip-mirip dengan tokoh utama di game *Grand Theft Auto*.

```

AKSI PENGEMUDI
Pilih jenis kendaraan : 3

Menjalankan Pesawat
Menghidupkan mesin terbang
Tarik handle
take off
terbang di angkasa
mendarat dengan selamat
Mematikan mesin terbang
  
```

Gambar 10.22

Tantangan

Sekarang giliran Anda. Buat sebuah tipe kendaraan baru bernama `MobilTerbang` dan `Sepeda`. Satu-satunya file yang boleh Anda rubah hanyalah `class FactoryKendaraan`. (Kelemahan dari menggunakan pola desain *simple factory* adalah pelanggaran terhadap prinsip OCP pada `class FactoryKendaraan`)

Petunjuk: `MobilTerbang` adalah jenis kendaraan bermotor yang memiliki dua mesin (komposisi), yaitu `MesinDarat` dan `MesinTerbang`. Sementara, `Sepeda` adalah kendaraan tidak bermotor (seperti becak).

Halaman ini sengaja dibiarkan kosong

Tentang Penulis



Mempelajari pemrograman secara otodidak sejak lulus SMA bukan perjalanan yang mudah bagi Nandi. Harus mencari materi-materi sendiri baik dari internet maupun dari buku-buku yang ada bukan perkara yang mudah di awal tahun 2000-an.

Sekarang, berbekal pengalaman dan ilmu yang diperoleh selama bekerja sebagai *software developer* di Jepang sejak tahun 2011, beliau berkomitmen untuk membantu Anda **memulai** belajar pemrograman dengan menulis buku-buku elektronik dan juga beberapa artikel yang bisa Anda dapatkan secara gratis di MahirKoding.id.

Beliau berharap karya-karyanya bisa menjawab kebingungan Anda ketika mencari materi-materi berbahasa Indonesia yang berkualitas dan mudah dimengerti.

Sampai buku ini ditulis, beliau masih tinggal di Jepang ditemani istri dan satu orang putranya.

Daftar Pustaka

1. Martin C, Robert. 2006. *Agile Principles, Pattern, and Practices in C#*. Boston: Prentice Hall.
2. Martin C, Robert. 2009. *Clean Code: A Handbook of Agile Software Craftmanship*. Boston: Prentice Hall.
3. Clark, Dan. 2013. *Beginning C# Object-Oriented Programming*. California: Apress.
4. Troelsen, Andrew dan Philip Japiske. 2017. *Pro C#7: With .NET and .NET Core*. California: Apress.
5. Skeet, Jon. 2019. *C# in Depth, Fourth Edition*. New York: Manning.
6. C# Guide. (n.d.). MSDN. <https://docs.microsoft.com/en-us/dotnet/csharp/>