Bagaiamana Berfikir Seperti Imuwan Komputer

Allen Downey
Diterjemahkan oleh Tim Buku Prodase - Universitas Telkom

Daftar Isi

Daftar Tabel

Daftar Gambar

Bab 1

Cara Program

Tujuan dari buku ini adalah mengajarkan kepada kamu bagaimana berfikir seperti seorang ilmuan komputer. Saya sangat suka bagaimana seorang ilmuan komputer berfikir karena mereka menggabungkan aspek-aspek terbaik dari ilmu Matematika, ilmu Teknik, dan ilmu pengetahuan alam. Sebagaimana halnya Matematikawan, ilmuan komputer menggunakan bahasa formal untuk menyatakan ide khususnya yang berkenaan dengan komputasi. Seperti Insinyur(*Engineer*), ilmuan komputer juga merancang sesuatu, merangkai beberapa komponen kedalam sebuah sistem dan mengeveluasi kelebihan dan kekurangan dari berbagai alternatif solusi. Mirip dengan ilmuan pada umumnya, para ilmuan komputer melakukan observasi tingkah laku dari sistem yang kompleks, membentuk beberap hipotesis dan menguji prediksi yang mereka buat.

Satu-satunya keahlian yang paling penting bagi seorang ilmuan komputer adalah keahlian dalam memecahkan masalah (*problem-solving*). Yang saya maksudkan dengan kemampuan memecahkan masalah adalah kemampuan mereka dalam melakukan formulasi masalah, berfikir secara kreatif mengenai solusi dari masalah yang telah diformulasikan dan mengekspresikan sebuah solusi secara jelas dan akurat. Dan ternyata

proses yang kamu lalui ketika belajar program komputer adalah sebuah kesempatan yang istimewa dalam berlatih keahlian memecahkan masalah. Itu kenapa judul dari bab ini adalah "Cara Program".

Pada satu sisi kamu akan belajar pemrograman, yang mana merupkah keahlian yang sangat penting. Disisi lainya, kamu akan menggunakan pemrograman sebagai sarana untuk belajar

1.1 Apa itu bahasa pemrograman?

Bahasa pemrograman yang akan kamu **p**elajari adalah Java, yang termasuk sebuah bahasa pemrograman yang relatif baru (Dirilis pertama kali oleh Sun Microsystem pada may 1995). Java adalah salah satu contoh dari bahasa pemrograman level tinggi (*high-level language*); bahasa pemrograman lain yang juga termasuk kategori bahasa pemrograman level tinggi adalah bahasa Python, C atau C++ dan Perl.

Selain istilah bahasa pemrograman level tinggi, terdapat juga istilah bahasa pemrograman level rendah (*low level languages*) dan terkadang dikenal juga dengan istilah bahasa mesin atau bahasa *assembly*. Pada kenyataanya, komputer hanya bisa memahami bahasa pemrograman level rendah. Oleh sebab itu, sebuah program yang ditulis menggunakan bahasa level tinggi harus diterjemahkan terlebih dahulu ke bentuk bahasa level rendah sebelum program tersebut dijalankan. Proses penterjemahan ini membutuhkan waktu sebelum bisa dijalankan oleh komputer, hal ini menjadi salah satu kekurangan dari bahasa pemrograman level tinggi.

Keunggulan dari bahasa level-tinggi cukup banyak jika dibandingkan dengan kekuranganna. Pertama, jauh lebih mudah untuk membuat program dengan menggunakan bahasa level-tinggi; waktu yang dibutuhkan untuk menuliskan program jauh lebih singkat, penulisannya juga jauh lebih pendek dan mudah dibaca jika dibandingkan dengan bahasa level-rendah. Keuntungan yang kedua adalah portabilitas dalam

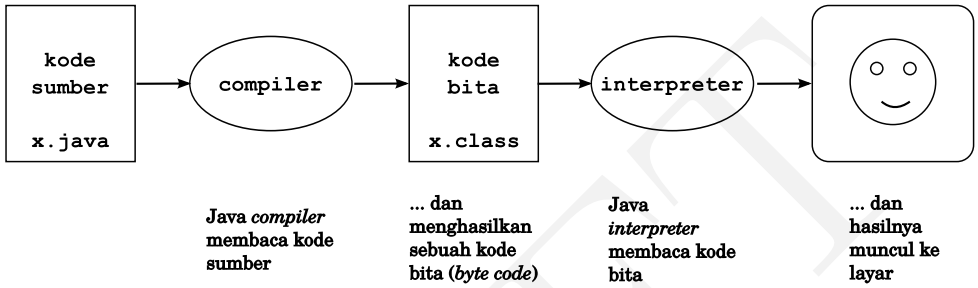
menjalankannya diberbagai macam arsitektur komputer dengan tanpa modifikasi. Berbeda halnya dengan program yang ditulis dengan bahasa level-rendah yang hanya bisa dijalankan di komputer tertentu, sehingga perlu dimodifikasi jika ingin dijalankan pada komputer dengan arsitektur yang berbeda.

Oleh karena kelebiha-kelebihan tersebut, maka hampir semua program ditulis dengan menggunakan bahasa pemrograman level-tinggi. Bahasa level-rendah hanya digunakan untuk membuat program-program tertentu saja yang jumlahnya juga sedikit.

Terdapat dua cara untuk menterjemahkan sebuah program; **interpretasi** (*interpreting*) dan **kompilasi** (*compiling*). Sebuah *interpreter* adalah sebuah program yang membaca sebuah program level-tinggi dan melakukan apa yang diminta oleh program tersebut. Sebagai akibatnya, interpreter akan menterjemahkan program baris demi baris Sementara *compiler* adalah sebuah program yang membaca sebuah program level-tinggi dan menterjemahkan keseluruhan program secara langsung, sebelum menjalankan perintah apa pun dari program tersebut. Sering kali, kamu akan melakukan proses kompilasi(*compiling*) secara terpisah terlebih dahulu, kemudian baru menjalankan (*run*) program. Pada kasus ini, program level-tinggi disebut dengan istilah kode sumber (*source code*) dan program yang telah diterjemahkan disebut dengan istilah kode objek (*object code*) atau *executable*.

Java adalah sebuah bahasa pemrograman yang menggunakan kompilasi dan juga interpretasi ketika menjalankan program. Alih-alih menterjemahkan program ke dalam bahasa mesin, *Java compiler* mengubahnya ke dalam bentuk kode bita (*byte code*). Sama halnya dengan bahasa mesin, kode bita sangat mudah dan juga cepat untuk diterjemahkan (interpretasi). Perbedaannya, kode bita tidak bergantung pada arsitektur komputer tertentu (lebih *portable*) seperti halnya bahasa level-tinggi. Artinya sebuah kode bita yang dihasilkan di sebuah komputer dapat dipindahkan dan dijalankan di komputer lain yang berbeda mesin/arsitektur. Kemampuan ini merupakan salah

satu kelebihan dari bahasa Java jika dibandingkan dengan bahasa level-tinggi lainnya.



Gambar 1.1: Proses kompilasi dan interpretasi pada bahasa Java

Walaupun proses ini terlihat kompleks, namun di beberapa perangkat lunak (*software*) yang digunakan untuk meprograman (sering disebut Integrated Development Environment/IDE) proses-proses tersebut telah dibuat otomatis untuk kamu. Sehingga kamu hanya cukup menekan sebuah tombol "run" saja, maka *software* IDE tersebut akan melakukan kompilasi dan interpretasi untuk program yang kamu buat.Namun disisi lain, kamu tetap harus tahu langkah-langkah yang terjadi dibalik proses yang telah terotomatis tadi, agar ketika terjadi kesalahan maka kamu dapat dengan mudah mengetahui penyebabnya.

1.2 Apa itu program?

Program adalah sebuah runtutan instruksi yang menyatakan bagaiman melakukan sebuah komputasi [1]. Istilah komputasi bisa berarti sebagai sesuati yang matematis, seperti menyelesaikan sebuah sistem persamaan atau menemukan akar dari sebuah polinomial, tetapi bisa

[1]Definisi ini tidak berlaku untuk seluruh bahasa pemrograman. Sebagai alternatif, lihat *http* : *//en.wikipedia.org/wiki/Declarative_programming*.

juga diartikan sebagai sebuah komputasi simbolik, seperti mencari dan mengganti teks pada sebuah dokumen atau mengkompilasi sebuah program.

Instruksi atau yang sering disebut dengan statemen (*statement*), memilik bentuk yang berbeda-beda untuk setiap bahasa pemrograman, namun terdapat beberapa instruksi dasar yang bisa dilakukan oleh seluruh bahasa pemrograman. instruksi-instruksi dasar tersebut antara lain:

1. **masukan(*input*):** instruksi-instruksi yang digunakan untuk mendapatkan data dari *keyboard* atau sebuah berkas atau dari perangkat lain

2. **keluaran (*output*):** instruksi-instruksi yang digunakan untuk menampilkan data kelayar atau mengirimkannya ke berkas atau perangkat lainnya

3. **matematika(*math*):** instruksi-instruksi yang digunakan untuk melakukan operasi matematika seperti penjumlahan, pengurangan, perkalian, pembagian dan lainya

4. **pengkondisian(*testing*):** instruksi-instruksi yang digunakan untuk memeriksa kondisi tertentu dan menjalankan urutan statemen yang sesuai.

5. **perulangan(*repetition*):** instruksi-instruksi yang digunakan untuk melakukan pengulangan terhadap sebuah atau beberapa statemen.

Setiap program yang pernah kamu gunakan, tidak peduli seberapa rumit apapun program tersebut, pasti tersusun dari kombinasi dari instruksi-instruksi dasar di atas. Oleh sebab itu, salah satu cara untuk menjelaskan pemrograman adalah sebagi sebuah proses yang memecah-mecah sebuah pekerjaan yang besar dan kompleks kedalam bentuk

beberapa sub pekerjaan yang jauh lebih kecil secara terus menerus hingga sub pekerjaan tersebut dapat secara sederhana dijalankan dengan menggunakan salah satu instruksi-instruksi dasar tadi.

1.3 Apa itu *debugging*?

Untuk alasan lelucon, error (keselahan) yang terdapat pada pemrograman disebut dengan "kutu" (*bug*) dan proses yang dilakukan untuk menemukan dan memperbaik "kutu" tersebut dikenal dengan istilah *debugging*.

Terdapat tiga jenis *error* yang sering muncul dalam sebuah program yaitu *syntax error*, *run-time error* dan *logic dan semantik error*. Penting dan sangat bermanfaat sekali bagi kamu jika kamu bisa membedakan ketiganya, sehingga kamu dapat dengan cepat dan juga mudah dalam menelusuri kesalahan yang ada dan kemudian memperbaikinya.

1.3.1 Kesalahan Sintak *(Syntax Error)*

Compiler hanya bisa melakukan kompilasi jika kode program yang ditulis telah benar secara sintak, jika tidak maka proses kompilasi akan gagal dan kamu tidak akan dapat menjalankan program tersebut. Sintak berarti struktur dan juga aturan dari struktur penulisan dari kode program yang kamu buat.

Sebagai contoh, dalam bahasa inggris sebuah kalimat harus dimulai dengan huruf besar dan diakhiri dengan tanda titik.

```
ini adalah contoh kalimat yang salah secara sintak.
Ini juga salah secara sintak
```

Bagi kebanyakan pembaca, sedikit kesalahan sintak bukanlah masalah yang berarti. Sebagai contoh *kta msh bsa mmbaca dan memhami tlsan ini dngn baik* walaupu penulisan kata-katanya banyak yang tidak lengkap.

Kemampuan seperti itu tidak dimiliki oleh *compiler*, jika terjadi satu satu kesalahan penulisan (sintak) di kode program yang kamu buat maka *compiler* akan menampilkan pesan error *error message* dan seketika itu akan menghentikan proses kompilasi. Ketika hal tersebut terjadi, maka program yang kamu buat bisa dipastikan tidak akan bisa dijalankan.

Lebih buruk lagi, dalam bahasa Java terdapat lebih banyak aturan sintak dibandingkan dengan sintak yang dimiliki oleh bahasa inggris atau bahasa lainnya, dan pesan error yang tampilkan sering sekali tidak terlalu membantu. Seiring bertambahnya pengalaman kamu, maka kamu akan membuat sedikit kesalahan dan akan lebih cepat dalam menemukannya.

1.3.2 Kesalahan ketika menjalankan *(Run-Time Error)*

Jenis kesalahan berikutnya adalah kesalahan yang muncul ketika menjalankan program (*Run-time Error*). Disebut seperti itu, karena kesalahan tersebut tidak muncul hingga program dijalankan. Di Java, sebuah *run-time error* terjadi ketika *interpreter* menjalankan kode bita (*byte code*) dan terjadi kesalahan dalam proses tersebut.

Java adalah sebuah bahasa pemrograman yang cendrung aman, artinya seluruh potensi kesalahan diupayakan dapat dideteksi oleh *compiler*. Sehingga *run-time error* dapat diminimalisir, khususnya untuk program-program yang sederhana.

Di Java, *run-time error* dikenal ddngan istilah *exceptions*(pengecualian). Dan di banyak lingkungan, *ecxeptions* muncul dalam bentuk *windows* atau *dialog box* yang berisi informasi mengenai apa yang telah terjadi dan apa yang program lakukan ketika hal tersebut terjadi. Informasi ini sangat berguna sekali ketika melakukan *debugging*

1.3.3 Kesalahan logika dan semantik *(Logic errors and sematics)*

Jenis *error* yang ketiga adalah kesalahan logika dan semantik. Jika program kamu memiliki kesalahan logika, maka program kamu akan dikompilasi dan dijalankan tanpa adanya *error message*, namun tidak akan melakukan apa yang seharusnya. Melainkan akan melakukan hal yang lain. Lebih spesifiknya, program yang kamu buat tidak akan melakukan apa yang kamu perintahkan.

Masalahnya adalah program yang kamu tulis bukanlah program yang kamu inginkan. Hal ini terjadi karena semantik atau makna dari kode program yang kamu buat salah. Menemukan kesalahan logika pada sebuah program adalah hal yang cukup rumit karena kamu harus bekerja secara terbalik, dimulai dari mengidentifikasi keluaran dari program dan mencoba untuk menganalisa apa yang sebenarnya terjadi.

1.3.4 Debugging

Salah satu keahlian yang sangat penting kamu kuasai pada pelajaran ini adalah *debugging*. Walaupun *debugging* bisa jadi membuat kamu frustasi, namun ia merupakan hal yang sangat menarik dan menantang dan merupakan bagian yang sangat bernilai dalam pemrograman.

Debugging mirip dengan pekerjaan detektif. Kamu berhadapan dengan petunjuk-petunjuk dan kamu harus menyimpulkan proses serta kejadian yang mengakibatkan hasil yang kamu lihat.

Debugging juga serupa dengan ilmu eksperimental (*experimental science*). Disaat kamu mendapati program yang kamu buat mengalami *error*, maka kamu akan membuat semacam dugaan (hipotesis) yang menjadi penyebab dari *error* tersebut, kemudian kamu memodifikasi kode program tersebut berdasarkan hipotesis yang kamu buat, dan kemudian kamu mencoba menjalankan ulang program kamu untuk membuktikan apakah hipotesis kamu benar/tidak. Jika hipotesis kamu

benar, maka kamu dapat memprediksi hasil dari modifikasi kamu, dan kamu semakin dekat dengan program yang berfungsi dengan baik. Sebaliknya, jika hipotesis kamu salah, maka kamu harus membuat hipotesis baru untuk kemudian mengulangi proses sebelumnya. Hal ini akan berlangsung secara terus menerus hingga *error* dari program kamu berhasil teratasi. Sebagaimana yang dikatakan oleh *Sherlock Holmes*, "Ketika kamu telah mengeliminasi hal-hal yang tidak mungkin, apa pun yang tersisa walaupun mustahil, haruslah sebuah kebenaran" (dari A. Conan Doyle's *The sign of Four.*)

Bagi sebagian orang, pemrograman dan *debugging* adalah hal yang sama. Karena pemrograman adalah sebuah proses *debugging* yang bertahap dari sebuah program hingga program terebut menjalankan apa yang kamu inginkan. Idenya adalah bahwa kamu harus memualai dengan sebuah program yang bisa dijalankan untuk melakukan sesuatu, tidak peduli sekecil apapun yang bisa dilakukan. Kemudian tambahkan modifikasi kecil ke program tersebut diikuti dengan *debugging*, sehingga dengan begitu kamu akan selalu memiliki sebuah program yang berjalan dengan baik dan program yang kamu buat makin lama akan semakin kompleks.

Sebagai contoh, Linux adalah sebuah sistem operasi yang terdiri dari ribuan baris kode program yang pada awalnya hanyalah sebuah program sederhana yang digunakan olen Linus Trovalds untuk mengeksplorasi *chip* Intel 80396. Menurut Larry Greenfield, " Salah satu proyek pertama yang dikerjakan ole Linus adalah sebuah program yang mencetak secara bergantian tulisan AAAA dan BBBB. Kemudian program ini berevolusi menjadi Linux" (dari *The Linux Users' Guide* Beta Version 1).

Pada bab berikutnya, saya menambahkan lebih banyak saran mengenai *debugging* dan praktik-prakatik pemrograman lainnya.

1.4 Bahasa Formal dan Bahasa Alami

Bahasa alami adalah bahasa-bahasa yang digunakan oleh orang-orang untuk berbicara, contohnya seperti bahasa Inggris,Spanyol dan Prancis. Bahasa tersebut tidak dirancang oleh manusia, namun berevolusi secara alami.

Bahasa formal adalah bahasa yang dirancang oleh manusia untuk aplikasi-aplikasi yang spesifik. Sebagai contoh,para matematikawan menggunakan bahasa formal dalam bentuk notasi-notasi tertentu untuk menyatakan hubungan antara bebara angka dan simbol. Pra kimiawan menggunakan bahasa formal untuk mereprsentasikan struktur kimiawi dari molekul-molekul. Dan yang paling penting adalah,

Bahasa pemrograman adalah bahasa formal yang telah dirancang untuk mengekspresikan komputasi

Bahasa formal memiliki aturan sintak yang ketat. Sebagai contoh, $3 + 3 = 6$ adalah pernyataan matematika yang benar secara sintak, sedangkan 3$ = merupakan pernyataan yang salah. Begitu halnya juga dengan H_2O adalah sebuah pernyataan nama zat kimia yang benar, namun tidak halnya untuk ${}_2Z_z$.

Aturan sintak terdiri dari dua hal yaitu aturan yang berkaitan dengan simbol(*token*) dan yang berkaitan dengan struktur. Toke merupakan elemen dasar dari sebuah bahasa, seperti halnya kata, angka, dan elemen kimiawi. Masalah yang terdapat pada pernyataan matematika 3$ = adalah $ merupakan bukan token yang legal dalam matematika (paling tidak sejauh yang saya tahu). Begitu halnya dengan ${}_2Z_z$, tidak legal karena tidak ada elemen kimia yang memiliki singkatan Z_z.

Aturan yang kedua adalah aturan yang berkenaan dengan struktur dari sebuah pernyataan, yaitu mengenai bagaimana token-token disusun. Pernyataan matematika 3$ = adalah pernyataan yang tidak legal secara struktur, karena kamu tidak dapat meletakkan tanda sama dengan (=) di akhir dari sebuah pernyataan matematis. Sama halnya

dengan penulisan ${}_2Z_z$ adalah tidak legal secara struktur, karena tidak boleh menempatkan *superscript* di posisi paling depan.

Ketika kamu membaca sebuah kalimat di bahasa Inggris atau pernyataak di bahasa formal, kamu harus mencari tahu atau memahami struktur dari kalimat tersebut (walaupun dalam bahasa alami kamu melakukannya secara tidak sadar). Proses seperti ini dikenal dengan istilah *parsing* (penguraian kalimat).

Walaupun bahasa alami dan bahasa formal memiliki kesamaan yang bersifat umum dalam hal token, struktur, sintak dan semantik, namun ada beberapa hal yang menjadi pembedanya, sebagaimana yang dijelaskan berikut;

keambiguan (*ambiguity*) : bahasa alami penuh dengan makna yang ambigu, yang biasanya ditangani dengan menggunakan petunjuk kontekstual dan informasi lainnya. Bahsa formal dirancang untuk tidak ambigu, yang artinya pernyataan apapun yang dituliskan dalam bahasa formal hanya memiliki satu makna saja tanpa melihat konteksnya.

pengulangan(*redundancy*) : Untuk mengatasi kalimat yang ambigu dan mengurangi kesalahpahaman, maka bahasa alami sering menggunakan pengulangan. Sedangkan bahasa formal jauh lebih ringkas.

kesastraan : Bahasa alami adalah bahasa yang sangat sarat denga idiom dan metafora, sementara bahasa formal lebih langsung kemakna sesungguhnya.

Bagi orang-orang yang tumbuh dan berinteraksi lama dengan bahasa alami seringkali mengalami kesulitan untuk menyesuaikan ke bahasa formal. Dalam beberapa hal, perbedaan antara bahasa alami dan bahasa formal mirip denga perbedaan antara puisi dan prosa.

Puisi Kata yang digunakan harus selaras antara bunyi dan makna yang diinginkan. Seluruh puisi secara keseluruhan membuat sebuah efek atau repon emosional. Keambiguan merupakan hal umum dan disengaja untuk membuat sebuah pusi.

Prosa Makna literal dari kata-kata jauh lebih penting dan struktur benar-benar berkontribusi terhadap makna yang diinginkan.

Program Makna dari sebuah program kamputer adalah tidak ambigu dan literal, serta dapat dipahami secara keseluruhan dengan analisa token dar sturkturnya.

Berikut ini beberapa saran untuk membaca program (dan bahasa formal lainnya). Pertama, kamu harus ingat bahwa bahasa formal jauh lebih padat daripada bahasa alami, sehingga membutuhkan waktu yang lama untuk membacanya. Sturuktur pada bahasa formal adalah hal yang sangat penting, dan bukanlah hal yang tepat jika membacanya dari atas ke bawah dan dari kiri ke kanan. Cobalah membacanya dengan belajar melakukan penguraian dari program di fikiran kamu, identifikasi token-tokenya dan terjemahkan strukturnya. Akhirnya, kamu perlu ingat bahwa detail amatlah penting dalam bahasa formal. Hal-hal kecil seperti kesalahan ejaan dan tanda baca yang biasanya kamu abaikan dalam bahasa alami, dapat menjadi perbedaan yang besar dalam bahasa formal.

1.5 Program Pertama

Secara tradisional, program pertama yang ditulis oleh orang yang baru belajar bahasa pemrograman adalah `"hello world"` karena program tersebut sederhana, hanya menampilkan tulisan `"hello world"` ke layar. Jika program tersebut ditulis dalam bahasa Java, maka bentuknya sebagai berikut;

```
1 public class Hello{
2
3    //main: menghasilkan beberapa keluaran sederhana
4
5    public static void main (String[] args){
6       System.out.println("Hello, world.");
7    }
8 }
```

Program ini mengandung beberapa hal yang cukup sulit jika dijelaskan ke para pemula.Namun program ini menyediakan preview dari topik-topik yang akan kita pelajari nanti.

Program java terdiri dari definisi class yang memiliki bentuk seperti berikut:

```
1
2 public class NAMAKELAS{
3
4    public static void main (String[] args){
5       PERNYATAAN
6    }
7
8 }
```

`NAMAKELAS` menunjukan sebuah nama kelas yang dipilih oleh programmer. Pada contoh sebelumnya, nama kelas yang dipilih adalah `Hello`.

`main` adalah sebuah *method*/fungsi yang merupakan kumpulan dari beberapa pernyataan. Nama `main` sendiri adalah khusus karena menandakan bagian pertama kali yang akan dijalankan dalam bahasa java. Ketika sebuah program dijalankan maka akan dimulai dari bagian `main` dan berakhir ketika program tersebut menyelesaikan pernyataan

terakhir yang terdapat pada method `main`.

Di dalam *method* `main` dapat terdiri dari beberapa buah pernyataan, namun pada contoh di atas hanya terdapat satu pernyataan saja yaitu pernyataan untuk mencetak (*print statement*) ke layar. Di Java setiap pernyataan selalu diakhiri dengan tada titik koma (;), begitu pula pada pernyataan untuk mencetak diatas.

`System.out.println` adalah sebuah *method* yang telah disediakan oleh salah satu pustaka(*library*) Java. Sebuah pustaka/*library* adalah kumpulan definisi dari Class dan method.

Java menggunakan kurung karawal ({ dan }) untuk mengelompokkan sesuatu secara bersama-sama. Kurung karawal terluar yang terdapat pada baris ke-1 dan ke-8 memuat definisi Class. Sementara kurung karawal yang terdalam berisi definisi dari *method* `main`.

Baris ke-3 yang dimulai dengan tanda gari miring dua kali (//) menandakan sebuah komentar, biasanya digunakan untuk menjelaskan kode program yang berada di bawahnya. Ketika *compiler* menjumpai tanda // di sebuah program maka baris kode tersebut akan diabaikan.

1.6 Glosarium

problem-solving: Proses yang dilakukan untuk memformulasikan sebuah masalah, mencari alternatif solusi kemudian mewujudkan solusi yang dipilih.

bahasa tingkat tinggi(*high-level language*): Sebuah bahasa pemrograman, seperti Java yang dirancang agar mudah dibaca dan ditulis oleh manusia.

bahasa tingkat rendah *low-level language*: Sebuah bahasa pemrograman yang dirancang agar mudah bagi komputer untuk menjalankannya. Dikenal juga dengan sebutan "bahasa mesin" atau "bahasa assembly".

bahasa formal: Bahasa apa pun yang dirancang untuk tujuan yang spesifik, seperti untuk menyatkan ide-ide matematis atau program komputer. Seluruh bahasa pemrograman adalah bahasa formal.

bahasa alami: Bahasa apa pun yang digunakan oleh manuasi untuk berbicara dan telah berevlusi sceara alami.

portabilitas: Kemampuan yang dimiliki oleh sebuah program untuk dijalankan debeberapa komputer yang berbeda.

interpretasi: Sebuah aktifitas yang dilkakuan untuk menterjemahkan sebuah program bahasa tingkat tinggi baris demi baris.

kompilasi: Sebuah aktifitas yang dilakukan untuk menterjemahkan sebuah program dari bahasa level tinggi ke dalam bentuk bahasa level rendah secara keseluruhan untuk dijalankan kemudian.

source code: Sebuah program dalam bahasa level tinggi, sebelum melewati proses kompilasi.

kode objek (*object code*) : Keluaran yang dihasilkan dari proses kompilasi, setelah menterjemahkan program.

***executable*:** Nama lain dari *object code*, yaitu sebuah program yang dapat dijalankan oleh komputer.

kode bita (*byte code*): Sejenis *object code* yang digunakan pada program java. *Byte code* serupa dengan bahasa level rendah namun ia memiliki portabilitas seperti bahasa level tinggi.

pernyataan(*statement*): Bagian dari sebuah program yang mengekspresikan sebuah komputasi.

***print statement*:** Sebuah pernyataan (*statement*) yang digunakan untuk menampilkan sebuah hasil ke layar.

komentar (*comment*): Bagian dari program yang mengandung informasi mengenai kode program dan tidak memiliki efek terhadap program ketika dijalankan.

***method*:** Sebuah sebutan untuk sekumpulan pernyataan (*statement*).

pustaka(*library*): Koleksi dari definisi Clas dan method.

***bug*:** Kesalahan yang terdapat pada program.

sintak: Struktur penulisan dari sebuah program.

semantik: Makna dari program

***parse*:** Memeriksa sebuah program dan menganalisa struktur sintaktis dari program tersebut.

kesalahan sintak (*syntax error*): Sebuah kesalahan struktur penulisan yang mengakibatkan sebuah program tidak bisa dikompilasi.

***exception*:** Sebuah kesalahan pada sebuah program yang mengakibatkan program terhenti ketika sedang dijalankan. Kesalahan jenis ini dikenal juga dengan istilah *run-time error*

kesalahan logika (*logic error*): Sebuah kesalahan pada sebuah program dimana program yang dibuat tidak menjalankan apa yang sebenarnya diminta.

debugging Sebuah proses yang dilakukan untuk menemukan dan kemudian menghilangkan sebuah kesalahan (tiga jenis kesalahan yang telah dibahas sebelumny) pada sebuah program.

1.7 Latihan

Bab 2

Membuat Objek

2.1 Definisi Class dan Tipe Objek

Kembali pada Bagian 1.5 ketika kita mendefinisikan kelas Hello, kita juga membuat jenis objek bernama Hello. Kita tidak menciptakan variabel dengan tipe Hello, dan kami tidak membuat objek Hello, tapi kita bisa buat !

Contoh yang tidak masuk akal, ketika tidak ada alasan untuk membuat objek Hello, dan itu tidak akan berbuat lebih jika kita lakukan. Dalam bab ini, kita akan melihat definisi Class yang menciptakan jenis objek yang berguna.

Berikut adalah hal yang paling penting dalam bab ini:

•Mendefinisikan Class baru juga menciptakan jenis objek baru dengan nama yang sama.

•Sebuah definisi Class seperti template untuk objek, menentukan apa variabel misalnya memiliki objek dan metode apa yang dapat beroperasi pada method.

•Setiap objek memiliki beberapa jenis objek.

•Ketika kamu membuat sebuah objek baru, Java memanggil method

khusus yang disebut constructor untuk menginisialisasi variabel instance. Kamu dapat memberikan satu atau lebih konstruktor sebagai bagian dari definisi kelas.

•Method yang dapat beroperasi pada tipe yang dapat didefinisikan dalam class tersebut.

Berikut adalah beberapa masalah tentang syntax pada definisi class:

•Nama Class, harus diawali dengan huruf kapital, yang membantu membedakan apakah itu tipe primitif dan nama variabel.

•Kamu biasanya menempatkan satu definisi kelas di setiap file, dan nama file harus sama dengan nama kelas, dengan .java akhir nama file. Sebagai contoh, Class Time didefinisikan dalam file bernama Time.java.

•Dalam program apapun, satu kelas yang ditunjuk sebagai class awal. Class awal harus berisi sebuah method main (Method Utama), yang mana program tersebut akan dieksekusi. Kelas-kelas lain mungkin memiliki method main, tapi itu tidak akan dieksekusi.

Dengan masalah tersebut. Mari lihat contoh pada Class Time(Waktu).

2.2 Waktu

Sebuah motivasi untuk menciptakan jenis objek adalah untuk merangkum data dalam suatu objek yang dapat dilakukan sebagai satu kesatuan. Kita telah melihat dua jenis seperti ini, titik dan kotak.

Contoh lain, yang kita akan menerapkan diri kita sendiri, adalah waktu, yang merepresentasikan waktu dalam hari. Data disimpan dalam sebuah objek Waktu adalah satu jam, satu menit, dan sejumlah detik. Karena setiap objek Waktu mengandung data jam, menit, dan detik, maka kita perlu variabel.

Langkah pertama adalah untuk menentukan jenis setiap variabel. Jelas bahwa jam dan menit harus bilangan bulat (int). Untuk lebih

menarik, maka kita buat detik menggunakan double.

Contoh variabel dideklarasikan di awal definisi class, seperti berikut

```
class Waktu {
int jam, menit;
double detik;
}
```

Dengan sendirinya, kode diatas merupakan definisi class. Setelah mendeklarasikan variabel, langkah selanjutnya yaitu menentukan konstruktor untuk class baru.

2.3 Konstruktor

Konstruktor adalah inisialisasi dari variabel. Sintaks untuk konstruktor mirip seperti method lain,tapi dengan tiga pengecualian:

- Nama konstruktor sama dengan nama class.
- Konstruktor tidak memiliki nilai kembali (return type).
- Keyword statis dihilangkan.

Kita berikan contoh untuk class Waktu:

```
public Waktu(){
this.jam = 0;
this.menit = 0;
this.detik = 0.0;
}
```

Mana yang kamu harapkan untuk melihat return type, antara public dan Waktu, tidak ada bukan ?. Itulah cara kita (dan compiler) dapat memberitahu bahwa ini adalah sebuah konstruktor.

Konstruktor tidak akan mengambil argumen. Setiap baris dari konstruktor menginisialisasi sebuah variabel untuk nilai default. Nama ini adalah kata kunci khusus yang mengacu pada objek yang kita ciptakan. Kamu dapat menggunakan ini dengan cara yang sama kamu gunakan untuk nama benda lainnya. Sebagai contoh, Kamu dapat membaca dan menulis variabel contoh ini, dan Kamu dapat melewati ini sebagai argumen untuk method lain.

Tapi kamu tidak menyatakan ini dan kamu tidak dapat membuat sebuah tugas untuk itu. ini dibuat oleh sistem, yang harus kamu lakukan adalah menginisialisasi variabel nya.

2.4 Konstruktor (Lanjutan)

Konstruktor dapat overload, seperti method lain, yang berarti bahwa kamu dapat memberikan beberapa konstruktor dengan parameter yang berbeda. Java tahu yang konstruktor dengan cara mencocokkan argumen baru dengan parameter konstruktor.

Umum untuk memiliki satu konstruktor yang tidak memiliki argumen, dan satu konstruktor yang mengambil daftar parameter identik dengan daftar variabel misalnya. Sebagai contoh:

```
public Waktu (int jam, int menit, double detik){
this.jam = jam;
this.menit = menit;
this.detik = detik;
}
```

Nama dan jenis parameter akan sama dengan nama dan jenis variabel. Semua konstruktor tidak menyalin informasi dari parameter untuk variabel.

Jika kamu melihat dokumentasi untuk point dan kotak, kamu akan melihat bahwa kedua kelas menyediakan konstruktor seperti ini. Over-

loading konstruktor menyediakan fleksibilitas untuk membuat objek pertama dan kemudian mengisi kekosongan, atau untuk mengumpulkan semua informasi sebelum membuat objek.

Ini mungkin tidak tampak menarik. Menulis konstruktor adalah membosankan, prosesnya selalu mekanis. Setelah kamu telah menulis dua variabel, Kamu akan menemukan bahwa kamu dapat menulis dengan cepat hanya dengan melihat daftar variabel misalnya.

2.5 Membuat Objek Baru

Meskipun konstruktor terlihat seperti method, Kamu tidak pernah memanggil mereka secara langsung. Sebaliknya, ketika kamu memanggil konstruktor baru, dimulai dari sistem mengalokasikan ruang untuk objek baru baru kemudian kamu akan memanggil konstruktor.

Program berikut menunjukkan dua cara untuk membuat dan menginisialisasi objek Waktu:

```
class Waktu {
int jam, menit;
double detik;

public Waktu(){
this.jam = 0;
this.menit = 0;
this.detik = 0.0;
}

public Waktu(int jam, int menit, double detik){
this.jam = jam;
this.menit = menit;
this.detik = detik;
}
```

```
public static void main(String[] args){
Waktu w1 = new Waktu();
w1.jam = 11;
w1.menit = 8;
w1.detik = 3.14159;
System.out.println(w1);

Waktu w2 = new Waktu(11, 8, 3.14159);
System.out.println(w2);
}
}
```

Dalam method main, pertama kali kita memanggil objek baru, kita tidak memberikan sebuah argumen, sehingga Java memanggil konstruktor pertama. Beberapa baris berikutnya baru memberikan nilai pada variabel.

Pada saat yang kedua kita panggil objek baru, kita memberikan argumen yang sesuai dengan parameter konstruktor kedua. Ini cara menginisialisasi variabel misalnya lebih ringkas dan sedikit lebih efisien, tetapi bisa sulit untuk dibaca, karena tidak jelas mana nilai yang ditugaskan untuk variabel misalnya.

2.6 Mencetak Objek

Hasil output pada program sebelumnya yaitu :
Waktu@80cc7c0
Waktu@80cc807

Ketika Java mencetak nilai dari user-defined, mencetak nama jenis dan heksadesimal khusus (basis 16) kode yang unik untuk setiap objek. Kode ini tidak berarti dalam dirinya sendiri; pada kenyataannya, itu

dapat bervariasi dari mesin ke mesin dan bahkan dari compiler untuk menjalankan. Tetapi dapat berguna untuk debugging, jika kamu ingin melacak objek individu.

Untuk mencetak objek dalam cara yang lebih berarti untuk pengguna, Kamu dapat menulis sebuah metode yang disebut sesuatu seperti cetakWaktu:

```
public static void cetakWaktu(Waktu w){
System.out.println(w.jam + ":" + w.menit + ":" + w.detik);
}
```

Output dari method ini, jika kita beranggapan baik w1 atau w2 sebagai argumen, outputnya adalah 11: 8: 3.14159. Meskipun ini dikenali sebagai waktu, tidak cukup dalam format standar. Sebagai contoh, jika jumlah menit atau detik kurang dari 10, kita mungkin ingin menghapus bagian desimal dari detik. Dengan kata lain, kita ingin sesuatu seperti 11:08:03.

Dalam kebanyakan bahasa, ada cara sederhana untuk mengontrol output format untuk nomor. Di Javaa tidak ada cara sederhana.

Java menyediakan sebuah mekanisme untuk mencetak hal-hal seperti format waktu dan tanggal, dan juga untuk menafsirkan masukan apa yang di format. Kamu dapat melihat dokumentasi untuk class Tanggal dalam paket java.util.

2.7 Operasi pada Objek

Dalam bagian berikutnya, terdapat 3 macam metode yang beroperasi pada objek :

•**Pure Function** : Mengambil objek sebagai argumen tetapi tidak dapat memodifikasi. Nilai return yaitu baik objek primitif atau objek baru dibuat di dalam metode.

•**Modifier** : Mengambil objek sebagai argumen dan dapat dimodifikasi beberapa atau semua yang terdapat pada modifier tersebut. bahkan

nilai return void.
•**Fill-in Method** : Salah satu objek yang kosong, Objek akan diisi oleh method itu sendiri. Secara teknis, fill-in method adalah tipe modifier.

2.8 Pure Functions

Sebuah method dianggap sebagai fungsi murni jika hasilnya bergantung hanya pada argumen, dan tidak memiliki efek lainnya seperti memodifikasi sebuah argumen atau mencetak sesuatu. Satu-satunya hasil fungsi murni adalah nilai kembali.

Salah satu contoh adalah isAfter, yang membandingkan dua Waktu dan mengembalikan tipe boolean yang menunjukkan apakah operan pertama datang setelah kedua:

```
public static boolean isAfter(Waktu w1, Waktu w2){
if (w1.jam >w2.jam)
return ture;
if (w1.jam <w2.jam)
return false;

if (w1.menit >w2.menit)
return true;
if (w1.menit <w2.menit)
return false;

if (w1.detik >w2.detik)
return false;
}
```

Contoh kedua adalah tambahWaktu, yang menghitung jumlah dua

kali. Sebagai contoh, jika 09:14:30, dan membuat roti membutuhkan 3 jam dan 35 menit, Kamu bisa menggunakan tambahWaktu untuk mencari tahu ketika roti akan dilakukan.

Berikut ini adalah draft kasar dari method ini yang tidak benar:

```
public static Waktu tambahWaktu(Waktu w1, Waktu w2){
Waktu tambah = new Waktu();
tambah.jam = w1.jam + w2.jam;
tambah.menit = w1.menit + w2.menit;
tambah.detik = w1.detik + w2.detik
return tambah;
}
```

Meskipun method ini mengembalikan sebuah objek Waktu, itu bukan sebuah konstruktor. Kamu harus kembali dan membandingkan sintaks dari method seperti ini dengan sintaks konstruktor, karena mudah untuk mendapatkannya.

Berikut adalah contoh bagaimana menggunakan method ini. Jika currentTime berisi waktu saat ini dan breadTime mengandung jumlah waktu yang diperlukan untuk membuat roti, maka kamu bisa menggunakan tambahWaktu untuk mencari tahu ketika roti akan dilakukan.

```
Waktu currentTime = new Waktu(9, 14, 30.0);
Waktu breadTime = new Waktu(3, 35, 0.0);
Waktu doneTime = tambahWaktu(currentTime, breadTime);
cetakWaktu(doneTime);
```

Output dari program ini adalah 12: 49: 30.0, yang benar. Di sisi lain, ada kasus di mana hasilnya tidak benar. Dapatkah Anda memikirkan yang tidak benar ?

Masalahnya adalah bahwa method ini tidak menangani kasus-kasus di mana jumlah detik atau menit menambahkan hingga lebih dari 60. Dalam hal ini, kita harus "membawa" detik-detik ekstra ke dalam

kolom menit, atau menit ekstra ke dalam kolom jam .

Berikut adalah versi method yang sudah dikoreksi.

public static Waktu tambahWaktu(Waktu w1, Waktu w2){
Waktu tambah = new Waktu();
tambah.jam = w1.jam + w2.jam;
tambah.menit = w1.menit + w2.menit;
tambah.detik = w1.detik + w2.detik;

if(tambah.detik >= 60.0){
tambah.detik -= 60;
tambah.menit += 1;
}

if (tambah.menit >= 60){
tambah.menit -= 60;
tambah.jam += 1;
}
}
Meskipun benar, ada cara alternatif yang jauh lebih pendek.

Kode ini menunjukkan dua operator, kita belum pernah melihat sebelumnya, + = dan - =. Operator ini menyediakan cara ringkas untuk kenaikan dan penurunan variabel. Mereka mirip dengan ++ dan –,
kecuali
(1) variabel yang bekerja yaitu int, dan
(2) jumlah selisih tidak harus 1. Pernyataan sum.second - = 60,0; setara dengan sum.second = sum.second - 60;

2.9 Modifiers

Sebagai contoh pada modifier, increment, yang menambahkan jumlah detiks ke objek Waktu. Sekali lagi, konsep dari method ini terlihat seperti berikut :

```
public static void increment(Waktu waktu, double detiks){
waktu.detik += detik2;

if(waktu.detik >= 60.0){
waktu.detik -= 60.0;
waktu.menit += 1;
}

if(waktu.menit >= 60){
waktu.menit -= 60;
waktu.jam += 1;
}
}
```

Baris pertama melakukan operasi dasar; sisanya berkaitan dengan kasus yang sama yang kita lihat sebelumnya.

Apakah method ini sudah benar? Apa yang terjadi jika argumen detiks jauh lebih besar dari 60? Dalam hal ini, tidak cukup untuk mengurangi 60 sekali; kita harus terus melakukannya sampai detiks di bawah 60. Kita dapat melakukan itu dengan mengganti pernyataan dengan pernyataan sementara:

```
public static void increment(Waktu waktu, double detiks){
waktu.detik += detiks;

while(waktu.detik >= 60.0){
waktu.detik -= 60.0;
waktu.menit += 1;
```

```
}

while(waktu.menit >= 60){
waktu.menit -= 60;
waktu.jam += 1;
}
}
```

Solusi ini benar, tetapi tidak sangat efisien. Dapatkah kamu memikirkan solusi yang tidak memerlukan iterasi?

2.10 Fill-in Methods

Untuk menciptakan objek baru setiap kali tambahIsiWaktu dipanggil, kita bisa meminta untuk memberikan sebuah objek di mana tambahIsiWaktu menyimpan hasilnya. Bandingkan dengan versi sebelumnya:

```
public static void tambahIsiWaktu(Waktu w1, Waktu w2, Waktu tambah){
tambah.jam = w1.jam + w2.jam;
tambah.menit = w1.menit + w2.menit;
tambah.detik = w1.detik + w2.detik;

if(tambah.detik >= 60.0){
tambah.detik -= 60.0;
tambah.menit += 1;
}

if(tambah.menit <= 60){
tambah.menit -= 60;
tambah.jam += 1;
}
```

}
Hasilnya disimpan dalam variabel tambah, sehingga void merupakan return typenya.

Modifiers dan fill-in method merupakan yang efisien karena mereka tidak perlu membuat objek baru. Tapi mereka lebih sulit untuk mengisolasi bagian dari program; dalam skala proyek besar mereka dapat menyebabkan kesalahan yang sulit untuk menemukan / mencari data.

Pure functions dapat membantu mengelola kompleksitas proyek-proyek besar, sebagian dengan membuat beberapa jenis kesalahan, dan karena hasil dari pure functions hanya bergantung pada parameter, mungkin ada cara lain untuk mempercepat mereka dengan menyimpan hasil sebelumnya dihitung.

2.11 Perencanaan dan Pengembangan pada Incremental

Dalam bab ini akan ditunjukkan proses pengembangan program yang disebut prototyping. Untuk setiap method, kalian akan menulis draft kasar yang dilakukan dalam perhitungan dasar, kemudian diuji pada beberapa kasus, kemudian mengoreksi apa yang kurang.

Pendekatan ini bisa efektif, tetapi dapat menyebabkan kode yang tidak terlalu rumit, karena berhubungan dengan banyak kasus khusus dan tidak dapat diandalkan karena sulit untuk meyakinkan diri sendiri bahwa kamu telah menemukan sebuah kesalahan.

Sebuah alternatif untuk mencari wawasan ke dalam suatu masalah yang dapat membuat pemrograman jauh lebih mudah. Dalam hal ini wawasan adalah bahwa waktu adalah benar-benar angka tiga digit dalam basis 60! Yang kedua adalah menit ”60 kolom”, dan jam adalah

"3600 kolom ."

Ketika kita menulis tambahWaktu dan increment, secara efektif dapat melakukan penambahan pada basis 60, itulah sebabnya mengapa kita harus " membawa dari satu kolom ke yang berikutnya.

Pendekatan lain untuk seluruh masalah adalah untuk mengkonversi Waktu menjadi double dan mengambil keuntungan dari fakta bahwa komputer sudah tahu bagaimana melakukan aritmatika dengan double. Berikut adalah metode yang mengubah waktu menjadi double:

```
public static double converttoSeconds(Waktu w){
int menit = w.jam * 60 + t.menit;
double detik = menit * 60 + w.detik;
return detik;
}
```

Sekarang semua yang kita butuhkan adalah cara untuk mengkonversi dari double untuk objek Waktu. Kita bisa menulis sebuah metode untuk melakukannya, tapi mungkin lebih masuk akal untuk menulis sebagai konstruktor:

```
public Waktu(double detik){
this.jam = (int)(detik / 3600.0);
detik -= this.jam * 3600.0;
this.menit = (int)(detik / 60.0);
detik -= this.menit * 60;
this.detiks = detik;
}
```

Konstruktor ini sedikit berbeda dari yang lain; melibatkan beberapa kalkulasi bersama untuk variabel.

Kamu mungkin harus berpikir untuk meyakinkan diri sendiri bahwa teknik yang digunakan untuk mengkonversi dari satu basis ke yang lain harus benar. Tapi setelah kamu yakin, kita dapat menggunakan metode ini untuk menulis ulang tambahWaktu:

```
public static Waktu tambahWaktu(Waktu w1, Waktu w2){
double detik = converttoSeconds(w1) + converttoSeconds(w2);
return new Waktu(detik);
}
```

Ini lebih pendek dari versi yang asli, dan jauh lebih mudah untuk menunjukkan bahwa itu adalah benar (dengan asumsi, seperti biasa, bahwa method yang dipanggil itu benar). Sebagai latihan, tulis ulang increment dengan cara yang sama.

2.12 Peraturan Umum

Dalam beberapa hal mengkonversi dari basis 60 ke basis 10 adalah hal yang lebih sulit daripada hanya berurusan dengan waktu. Konversi basis lebih abstrak; intuisi kita untuk menangani waktu harus lebih baik.

Tetapi jika kita memiliki wawasan untuk mengakali sebagai basis 60 angka, dan membuat investasi menulis metode konversi (convertToSeconds dan konstruktor pihak ketiga), kita mendapatkan sebuah program yang lebih pendek, lebih mudah dibaca dan dapat debug, dan lebih handal.

Hal ini juga lebih mudah untuk menambahkan fitur. Bayangkan jika kita kurangi dua kali untuk mencari durasi antara dua kali tersebut. Menggunakan metode konversi akan jauh lebih mudah.

Terkadang jika masalah lebih sulit membuat lebih mudah (kasus khusus yang lebih sedikit, lebih sedikit kesempatan untuk salah).

2.13 Algoritma

Ketika kamu menulis sebuah solusi umum untuk masalah dalam sebuah class, maka akan muncul solusi yang spesifik yang mengacu pada satu masalah, kamu akan menuliskannya pada sebuah algoritma.

Pertama, pertimbangkan beberapa hal yang bukan algoritma. Ketika Kamu belajar untuk memperbanyak nomor satu digit, Kamu mungkin menghafal tabel perkalian. Akibatnya, Kamu hafal 100 solusi spesifik, sehingga pengetahuanmu bukan secara algoritmik.

Tetapi jika kamu malas, kamu mungkin belajar beberapa trik. Misalnya, untuk produk dari n dan 9, Kamu dapat menulis n - 1 sebagai digit pertama dan 10 - n sebagai digit kedua. Trik ini adalah solusi umum untuk mengalikan nomor satu digit dengan 9. Itu adalah suatu algoritma!

Demikian pula, teknik yang kamu pelajari untuk penambahan dengan membawa, pengurangan dengan meminjam. Salah satu karakteristiknya algoritma adalah bahwa algoritma tidak memerlukan kecerdasan apapun untuk melaksanakannya. Algoritma adalah Sebuah instruksi untuk menyelesaikan sebuah masalah dalam class oleh proses mekanik.

Menurut pendapat saya, memalukan bahwa siswa menghabiskan begitu banyak waktu di sekolah belajar untuk mengeksekusi algoritma. Di sisi lain, proses merancang algoritma yang menarik, intelektual yang menantang, dan bagian sentral dari apa yang kita sebut pemrograman.

Beberapa hal yang dilakukan orang secara alami, tanpa kesulitan adalah yang paling sulit untuk mengekspresikan algoritma. Memahami bahasa alami adalah contoh yang baik. Kita semua melakukannya, tapi sejauh ini belum ada yang mampu menjelaskan bagaimana kita melakukannya, setidaknya tidak dalam bentuk algoritma.

Segera Kamu akan memiliki kesempatan untuk merancang algoritma sederhana untuk berbagai masalah.

2.14 Istilah - Istilah

•**Class** : Sebelumnya, sudah dijelaskan bahwa class kumpulan dari beberapa method. Dalam Bab ini kita belajar bahwa definisi class yaitu sebagai template (klise) untuk tipe objek yang baru.

•**Instance** : Sebuah member dalam class. Setiap objek pada instance merujuk pada beberapa class.

•**Konstruktor** : Method spesial yang menginisialisasikan variabel instance pada sebuah objek konstruktor yang baru.

•**Class Awal** : Class yang berisi method utama dimana program tersebut akan dieksekusi.

•**Pure Function** : Method yang hasilnya bergantung pada parameter dan tidak memiliki efek lain selain pengembalian nilai.

•**Modifier** : Method yang dapat mengganti satu atau beberapa objek yang dapat menerima parameter dan kebanyakan mengembalikkan nilai kosong.

•**Fill-in Method** : Tipe Method yang mengambil objek kosong sebagai parameter dan diisi sebagai variabel instance sebagai contoh dapat pengembalian nilai.

•**Algoritma** : Sebuah instruksi untuk menyelesaikan sebuah masalah dalam class oleh proses mekanik.

2.15 Latihan

Latihan 11.1. Dalam permainan papan Scrabble, setiap kotak berisi huruf, yang digunakan untuk mengeja kata-kata, dan nilai, yang digunakan untuk menentukan nilai dari kata-kata.
1. Tulis definisi untuk kelas bernama Ubin yang mewakili ubin pada Scrabble.
Variabel harus karakter berhuruf dan integer.

2. Tulis konstruktor yang mengambil parameter berhuruf , nilai dan menginisialisasi pada variabel.

3. Tulis method bernama printTile yang mengambil objek Tile sebagai parameter dan mencetak variabel.

4. Tulis method bernama testTile yang menciptakan objek Tile dengan huruf Z dan nilai 10, kemudian mgunakan printTile untuk mencetak hasil dari objek.

Inti dari latihan ini adalah untuk melatih pada bagian mekanismenya untuk menciptakan kelas baru dan kode yang tes tersebut.

Latihan 11.2. Tulis definisi kelas untuk Tanggal, jenis objek yang membandingkan tiga bilangan bulat, tahun, bulan dan hari. Kelas ini harus menyediakan dua konstruktor. Yang pertama harus ada parameter. Kedua harus mengambil parameter bernama tahun, bulan dan hari, dan menggunakannya untuk menginisialisasi variabel.

Tulis method utama yang menciptakan objek Tanggal bernama ulangTahun. Objek baru harus berisi tanggal lahir kamu. Kamu dapat menggunakan salah satu konstruktor.

Latihan 11.3. Bilangan rasional adalah angka yang dapat direpresentasikan sebagai rasio dua bilangan bulat. Misalnya, 2/3 adalah bilangan rasional, dan Kamu dapat memikirkan 7 sebagai bilangan rasional dengan implisit 1 di penyebut. Untuk tugas ini, Kamus akan menulis sebuah definisi kelas untuk bilangan rasional.

1. Buat program baru yang disebut Rational.java yang mendefinisikan kelas bernama Rational. Sebuah objek Rational harus memiliki dua contoh bilangan bulat (integer) untuk menyimpan pembilang dan penyebut.

2. Tulis konstruktor yang tidak mengambil argumen dan yang menentukan dari pembilang ke 0 dan penyebut untuk 1.

3. Tulis method printRational yang mengambil benda Rasional sebagai argumen dan mencetaknya.

4. Tulis method main yang menciptakan objek baru dengan tipe Rasional, dan membuat variabel misalnya untuk beberapa nilai, dan mencetak objek.

5. Pada tahap ini, Kamu sudah memiliki program yang dapat diuji. Ujilah dan jika perlu, debug itu.

6. Tulis konstruktor yang kedua untuk kelas kamu yang membutuhkan dua argumen dan gunakan untuk menginisialisasi variabel.

7. Tulis method yang disebut negate yang mengembalikkan bilangan rasional. Method ini harus modifier. Tambahkan baris main untuk menguji method baru.

8. Tulis method yang disebut invert yang mengembalikkan nomor

dengan swapping pembilang dan penyebut. Tambahkan baris main untuk menguji method baru.

9. Tulis method yang disebut toDouble yang mengubah bilangan rasional menjadi bilangan decimal (double) dan mengembalikan hasilnya. Method ini adalah pure functions, tidak dapat memodifikasi objek. Seperti biasa, uji method tersebut.

10. Tulis modifier bernama reduce yang mengurangi jumlah bilangan rasional untuk persyaratan terendah dengan mencari pembagi terbesar dari pembilang dan penyebut. Method ini harus menjadi pure functions; tidak harus memodifikasi variabel dari objek di mana ia dipanggil.

11. Tulis method yang disebut add yang mengambil dua angka Rational sebagai argumen dan mengembalikan sebuah objek Rational baru. return objek harus berisi jumlah dari argumen tersebut.

Ada beberapa cara untuk menambahkan fraksi. Kamu dapat menggunakan salah satu yang kamu inginkan, tetapi kamu harus memastikan bahwa hasil dari operasi berkurang sehingga pembilang dan penyebut tidak memiliki pembagi umum (selain 1).

Tujuan dari latihan ini adalah untuk menulis sebuah definisi kelas yang meliputi berbagai method, termasuk konstruktor, modifiers dan pure functions.