Pemrograman Berorientasi Objek

Edisi Pertama

Irfan Subakti

JALUR JAVA

Profitina
profitina.com

DAFTAR ISI

Kata Pengantar

Buku ini tumbuh dari mata kuliah Pemrograman Berorientasi Objek yang saya ampu di Departemen Teknik Informatika, Institut Teknologi Sepuluh Nopember (ITS) Surabaya, dan disusun sebagaimana kuliahnya disusun: 16 bab, masing-masing satu sesi yang bisa Anda ikuti, kerjakan, lalu pakai. Ini adalah edisi Java; edisi pendampingnya membahas 16 bab yang sama dalam bahasa pemrograman lain.

Sebelum menulis satu class pun, Anda perlu melihat sendiri masalah yang membuat pemrograman berorientasi objek diciptakan. Buku ini membangun kasus itu dengan sistem perpustakaan kecil — Pustaka — yang versi proseduralnya mulai kewalahan seiring bertumbuh, lalu membangunnya ulang, konsep demi konsep, pertama dengan Java lalu dengan Python.

Pemrograman Berorientasi Objek (OOP) hadir sebagai dua edisi paralel — satu dengan Java, satu dengan Python — yang mengajarkan roadmap 16-langkah yang persis sama, hanya berbeda pada idiom bahasanya, sehingga pembaca yang sudah menguasai satu edisi bisa memakai edisi lainnya sebagai perbandingan langsung bagaimana gagasan yang sama tampil di tiap bahasa. Kuliah ini dibuka dengan menunjukkan mengapa program manajemen perpustakaan prosedural (Pustaka) mulai kewalahan seiring bertumbuh, lalu membangunnya ulang sebagai class dengan field, constructor, dan enkapsulasi; menambahkan array/list dan pembentukan string; membahas exception handling dan file I/O; memperkenalkan interface dan abstract class; lalu masuk ke inheritance, polymorphism, generics dan collections framework, pemrograman GUI dengan event dan MVC, konkurensi (thread, race condition, sinkronisasi), dan jaringan dengan socket. Buku ini ditutup dengan menamai, di bab Design Patterns, apa yang sebenarnya sudah dilakukan kode Pustaka sejak lama — dan berakhir dengan proyek puncak client-server GUI-socket.

Setibanya Anda di halaman terakhir, Anda semestinya mampu:

- Mengenali kapan dan mengapa program prosedural perlu menjadi berorientasi objek — bukan sekadar cara menulis sintaks class.
- Merancang class di sekitar field, constructor, dan enkapsulasi, dalam idiom Java maupun Python.
- Memakai inheritance, polymorphism, interface/abstract class, dan generics untuk membangun kode yang fleksibel dan dapat dipakai ulang.
- Menangani exception dan file I/O secara andal, serta membangun antarmuka grafis dengan struktur event-driven MVC.
- Menulis kode konkuren secara aman — thread, race condition, sinkronisasi — serta kode jaringan dengan socket, client, dan server.
- Mengenali design pattern umum sebagai nama untuk struktur yang sudah dituju sendiri oleh kode Anda.

Ketika buku ini perlu menunjuk sistem nyata, ia menunjuk Profitina — profitina.com dan app.profitina.com — platform yang saya rancang, bangun, dan jalankan sendiri. Pilihan itu praktis, bukan promosi: saya bisa menunjukkan keputusan yang sesungguhnya, trade-off yang sesungguhnya, dan akibat yang sesungguhnya, karena sayalah yang harus menanggungnya. Contoh-contoh itu bukan analogi yang dicocok-cocokkan; di mana ia muncul, ia adalah persoalan yang sama, dilihat dari sudut pandang pengajaran.

Untuk siapa buku ini ditulis: mahasiswa yang sudah mengambil kuliah Dasar Pemrograman atau Struktur Data dan bisa menulis kode prosedural, dan kini butuh desain berorientasi objek — dalam Java, Python, atau keduanya.

Sepatah kata tentang metode. Setiap contoh kode dalam buku ini dikompilasi dan dijalankan sebelum dituliskan, dan lampirannya memuat catatan verifikasi sebagai buktinya. Ketika sebuah angka muncul, angka itu berasal dari pengukuran, bukan dari ingatan. Bila Anda menemukan kekeliruan, atau bagian yang bisa lebih jelas, saya sungguh ingin mendengarnya — begitulah edisi berikutnya menjadi lebih baik.

Surabaya, September 2026

Irfan Subakti
yifana@gmail.com

Profitina
profitina.com

BAB 1

Pengantar OOP: Dari Prosedur menuju Objek

Sebelum menulis satu class pun, Anda perlu gambaran jelas tentang masalah yang ingin dipecahkan oleh pemrograman berorientasi objek — dan mengapa "tambah saja lebih banyak fungsi" pada akhirnya berhenti bekerja.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menjelaskan, dengan contoh konkret, mengapa program prosedural berskala besar menjadi sulit diubah dengan aman. (2) Mendefinisikan objek sebagai gabungan data (field) dan perilaku (method), serta menjelaskan enkapsulasi sebagai "menyembunyikan data di balik perilaku." (3) Menyebutkan empat pilar OOP (enkapsulasi, pewarisan, polimorfisme, abstraksi) dan pratinjau di mana masing-masing diajarkan di mata kuliah ini. (4) Memasang dan memverifikasi Java Development Kit (JDK) dan IDE yang berfungsi. (5) Membaca dan menjelaskan setiap baris class Java minimal beserta program driver-nya, lalu meng-compile dan menjalankannya sendiri.

Satu Toolchain, Tiga Sistem Operasi — Windows 11, macOS, Ubuntu

OS	Instalasi sekali saja (JDK 21 LTS)	Kompilasi & jalankan (identik di mana pun)
Windows 11	winget install EclipseAdoptium.Temurin.21.JDK	javac Main.java lalu java Main
macOS	brew install --cask temurin@21 (atau installer adoptium.net)	javac Main.java lalu java Main
Ubuntu/Linux	sudo apt install openjdk-21-jdk	javac Main.java lalu java Main

Semua contoh di buku ini berjalan identik di Windows 11, macOS, dan Ubuntu/Linux; perintah hanya berbeda saat instalasi. Pilih baris Anda sekali dan ikuti terus di laptop Anda sendiri.

1.1 Selamat Datang di Pemrograman Berorientasi Objek

Selamat datang di mata kuliah Pemrograman Berorientasi Objek (OOP). Jika Anda sudah mengambil mata kuliah Dasar Pemrograman atau Struktur Data, Anda sudah tahu cara menulis fungsi, loop, array, dan kondisional — perangkat prosedural. Mata kuliah ini tidak membuang perangkat itu. Sebaliknya, mata kuliah ini mengajarkan cara berbeda mengorganisasi perangkat yang sama: alih-alih menyebar data ke berbagai array lepas dan mengoper data tersebut antar-fungsi yang berdiri sendiri, Anda akan belajar menyatukan data dan fungsi yang mengoperasikannya ke dalam satu unit bernama objek. Satu ide tunggal itu — data dan perilaku, disatukan, menyembunyikan detail internalnya sendiri — adalah benih tempat setiap konsep lain di mata kuliah ini tumbuh: enkapsulasi, pewarisan, polimorfisme, interface, generics, dan akhirnya design pattern yang dipakai di seluruh perangkat lunak produksi nyata, termasuk platform yang namanya tertera di sampul buku ini.

Mata kuliah ini mengajarkan setiap konsep dua kali: sekali dalam Java (bahasa OOP klasik, bertipe-statis, standar industri yang selalu dipakai mata kuliah ini) dan sekali dalam Python (bahasa bertipe-dinamis yang fitur OOP-nya lebih ringan namun secara konsep identik). Ini adalah edisi Java. Edisi Python pendamping memiliki peta jalan 16-lecture yang persis sama dan contoh kerja yang persis sama, diterjemahkan ke Python yang idiomatis — sehingga bahasa apa pun yang kelak dibutuhkan tim, pemberi kerja, atau proyek pribadi Anda, Anda sudah memahami ide dasarnya dan hanya perlu mempelajari sintaks baru, bukan konsep baru (Gambar 1.1).

Gambar 1.1 — Peta jalan 16-lecture OOP. Jalur Java dan Python sama-sama mengunjungi 16 pemberhentian yang persis sama; hanya idiom kode yang berbeda. Kita di sini: Lecture 1.

1.2 Mengapa Program Prosedural Melampaui Batasnya Sendiri

Bayangkan Anda membangun sistem perpustakaan kecil untuk ruang baca kampus — sebut saja Pustaka. Dalam gaya prosedural murni, Anda mungkin menyimpan koleksi sebagai beberapa array paralel: satu array judul, satu array penulis, satu array ISBN, dan satu array boolean yang mencatat apakah tiap buku sedang dipinjam. Setiap operasi — meminjam buku, mengembalikannya, memperpanjang, mencetak katalog — adalah fungsi bebas yang menerima keempat array tersebut sebagai parameter beserta indeks yang menunjukkan buku mana yang disentuh.

Ini berhasil, pada awalnya. Masalah muncul seiring program membesar. Tidak ada apa pun dalam bahasa yang mencegah fungsi mana pun, di mana pun dalam program seribu baris, untuk langsung menjangkau array onLoan dan mengubah satu entri menjadi true tanpa pernah memeriksa apakah buku itu sebenarnya tersedia, atau lupa memperbarui timesRenewed saat peminjaman diperpanjang, atau memperbarui satu array tapi tidak tiga array paralel saudaranya setelah bug diperbaiki hanya di satu fungsi. Tidak ada satu tempat pun dalam kode yang menjamin "data sebuah buku hanya bisa berubah dengan cara yang masuk akal untuk buku nyata" — tanggung jawab itu tersebar di setiap fungsi yang kebetulan menyentuh array-array tersebut, dan setiap fungsi baru yang ditambahkan ke program adalah satu tempat lagi di mana jaminan itu bisa diam-diam rusak.

> **"Yang penting hati-hati" tidak bisa diandalkan pada skala besar**
>
> Reaksi pertama yang umum adalah: "programmer-nya saja yang perlu hati-hati agar tidak menyalahgunakan array." Ini berhasil untuk tugas mahasiswa 200 baris yang ditulis satu orang dalam satu waktu duduk. Ini nyaris selalu gagal untuk program yang dirawat lebih dari satu orang, disentuh lebih dari sekali, atau lebih besar dari beberapa ratus baris — yang menggambarkan hampir semua perangkat lunak nyata yang dipakai di dunia kerja. OOP tidak membuat kesalahan ceroboh menjadi mustahil, tetapi ia memberi bahasa itu sendiri alat untuk membuat banyak kelas kesalahan mustahil untuk di-compile, bukan sekadar meminta programmer mengingat untuk tidak melakukannya.

1.3 Objek: Menyatukan Data dan Perilaku

Gerakan inti pemrograman berorientasi objek adalah berhenti memperlakukan data sebuah buku (judul, penulis, ISBN, status pinjam) dan perilaku sebuah buku (pinjam, kembalikan, perpanjang) sebagai hal terpisah yang tersebar di seluruh program. Sebaliknya, sebuah class mendefinisikan cetak

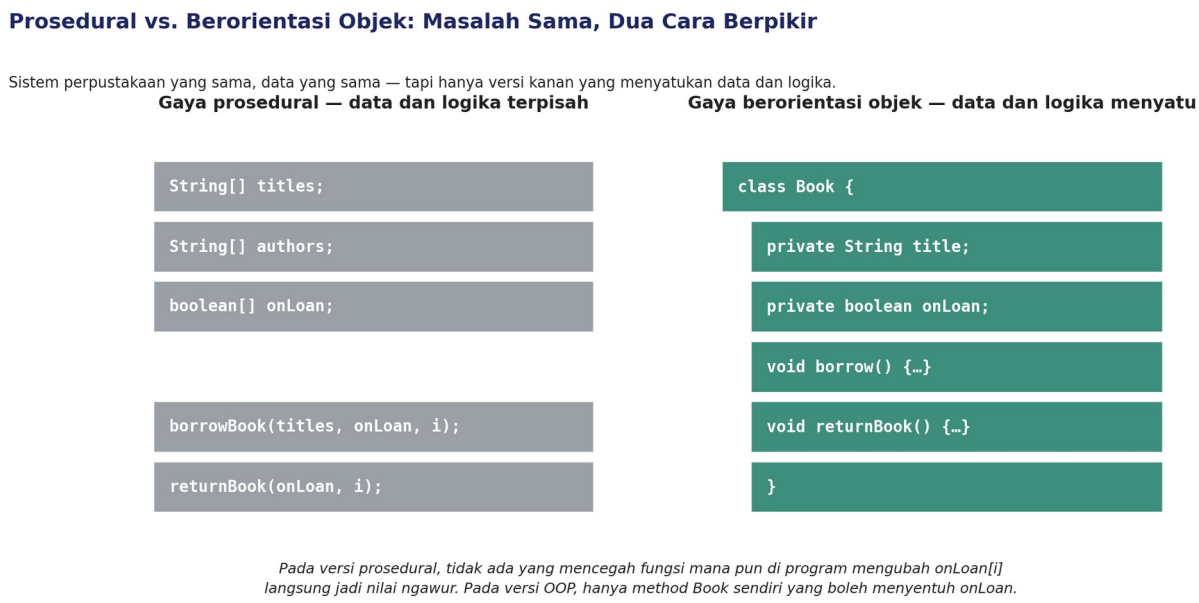
biru — Book — yang menyatukan field data dan method perilaku menjadi satu unit, dan sebuah objek adalah satu instans konkret dari cetak biru tersebut, persis seperti satu buku fisik di rak adalah satu instans konkret dari ide umum "sebuah buku".

Gambar 1.2 — Buku nyata dan objek Book menyimpan informasi yang sama; objeknya tambahan hanya memperlihatkan operasi (borrow, returnBook, renew) yang masuk akal dilakukan padanya.

> **Enkapsulasi, pratinjau**
>
> Menandai sebuah field sebagai private (seperti setiap field pada objek Book di Gambar 1.2) berarti tidak ada kode di luar class yang bisa membaca atau menulis field tersebut secara langsung — satu-satunya cara memengaruhi onLoan adalah memanggil borrow(), returnBook(), atau renew(), dan method-method itu bisa menolak permintaan yang tidak masuk akal (misalnya, meminjam buku yang sudah dipinjam). Penyatuan-plus-penyembunyian ini disebut enkapsulasi, dan ia adalah pilar pertama dari empat pilar OOP yang dibahas mata kuliah ini: enkapsulasi (Bab 2), pewarisan (Bab 9), polimorfisme (juga Bab 9), dan abstraksi lewat interface serta abstract class (Bab 7).

1.4 Prosedural vs. Berorientasi Objek: Masalah Sama, Dua Cara Berpikir

Gambar 1.3 menempatkan kedua gaya berdampingan untuk masalah perpustakaan Pustaka yang persis sama. Versi prosedural di kiri menyimpan empat array paralel dan sekumpulan fungsi bebas yang harus selalu diberi array yang benar dan indeks yang benar setiap kali. Versi berorientasi objek di kanan mendefinisikan satu class Book; setiap objek Book memiliki title dan flag onLoan-nya sendiri secara privat, dan satu-satunya cara mengubah onLoan adalah memanggil method yang didefinisikan dan bisa ditolak oleh Book itu sendiri.



Gambar 1.3 — Versi prosedural bisa dirusak dari mana pun dalam program. Versi berorientasi objek melindungi datanya sendiri lewat konstruksi, bukan sekadar konvensi.

Tidak ada satu gaya pun yang bisa melakukan sesuatu yang secara fundamental tidak bisa dilakukan gaya lainnya — OOP bukan jenis komputasi baru, melainkan cara berbeda mengorganisasi komputasi yang sama demi kejelasan, keamanan, dan ketahanan terhadap perubahan. Seiring program tumbuh melewati beberapa ratus baris dan melewati satu penulis saja, perbedaan organisasional itu menjadi pembeda antara basis kode yang bisa Anda perluas dengan aman selama bertahun-tahun dan basis kode di mana setiap perubahan berisiko merusak sesuatu yang jauh yang tidak pernah Anda sadari bergantung pada data lama yang tidak terlindungi.

1.5 Sekilas Empat Pilar

Mata kuliah ini disusun mengelilingi empat ide yang terus-menerus muncul dalam kode Java dan Python profesional. Bagian ini memberi pratinjau masing-masing dalam satu kalimat; masing-masing akan mendapat bab penuh nanti.

- Enkapsulasi (Bab 2) — menyatukan data dengan method yang mengoperasikannya, dan menyembunyikan data di balik method tersebut sehingga hanya bisa berubah dengan cara yang valid.
- Abstraksi lewat interface & abstract class (Bab 7) — mendeskripsikan APA yang bisa dilakukan sebuah objek ("ini bisa dipinjam") tanpa berkomitmen pada BAGAIMANA persisnya, sehingga jenis objek berbeda bisa dipakai secara bergantian di mana pun perilaku itu diharapkan.
- Pewarisan (Bab 9) — membiarkan satu class memakai ulang dan mengkhususkan field serta method class lain, sehingga ReferenceBook dan Textbook bisa berbagi hampir semua yang sudah didefinisikan Book umum.
- Polimorfisme (Bab 9) — memanggil nama method yang sama pada objek dari tipe konkret berbeda dan mendapatkan perilaku yang sesuai untuk tiap tipe, tanpa pemanggil perlu tahu tipe apa yang sedang dihadapinya.

Mengapa urutan ini?

Setiap basis kode Java atau Python profesional yang akan Anda baca bersandar pada keempat pilar secara bersamaan, sehingga urutan sebuah mata kuliah memperkenalkannya adalah pilihan pengajaran, bukan fakta tentang bahasa. Mata kuliah ini mengajarkan enkapsulasi lebih dulu karena ketiga pilar lainnya, dalam arti nyata, dibangun di atasnya: Anda tidak bisa membicarakan secara bermakna tentang mengkhususkan perilaku objek (pewarisan/polimorfisme) atau mendeskripsikan perilaku secara abstrak (interface) sebelum lebih dulu menerima bahwa data sebuah objek seharusnya disembunyikan di balik method-nya sendiri.

1.6 Menyiapkan Lingkungan Pengembangan Java Anda

Setiap contoh dalam buku ini ditulis untuk rilis Long-Term-Support (LTS) Java yang terkini. Per Agustus 2026, peta jalan resmi Oracle mencantumkan Java 8, 11, 17, 21, dan 25 sebagai rilis LTS, dengan Java 25 (rilis umum September 2025) sebagai yang terbaru — Oracle menyatakan berniat merilis LTS baru setiap dua tahun ke depan, dengan Java 29 direncanakan September 2027. Kode buku ini sendiri ditulis, di-compile, dan dijalankan pada JDK 21 LTS; semua yang ditampilkan bisa di-compile dan dijalankan tanpa perubahan pada JDK 25 juga, karena tidak ada materi mata kuliah ini yang bergantung pada fitur yang dihapus atau berubah di antara kedua rilis tersebut. Jangan memasang JDK 8 untuk mengikuti mata kuliah ini, meskipun Anda menemukan tutorial lama yang mengasumsikannya — JDK 8 sudah berhenti menerima update publik gratis bertahun-tahun lalu, dan beberapa API yang dipakai buku ini (record, pattern matching untuk switch, sealed class yang dirujuk di bab-bab berikutnya) tidak ada atau berperilaku berbeda padanya.

Alat	Rekomendasi	Alasan
JDK	Eclipse Temurin (Adoptium) atau Oracle JDK, versi 21 atau 25 LTS	Build OpenJDK gratis dan siap-produksi; kedua jalur LTS menerima update selama bertahun-tahun, tidak seperti rilis jangka pendek.
IDE — utama	IntelliJ IDEA Community Edition (gratis)	IDE Java paling banyak dipakai di industri saat ini; alat refactoring unggul, dukungan Git bawaan, dan alur setup proyek yang lebih ramah dibanding Eclipse untuk mata kuliah pertama.
IDE — alternatif	Eclipse IDE for Java Developers	Materi asli mata kuliah ini memakai Eclipse; tetap menjadi pilihan gratis yang sepenuhnya mumpuni dan banyak diajarkan bila lab atau kampus Anda sudah menstandarkannya.
Build tool (bab lanjutan)	Maven atau Gradle	Diperkenalkan saat sebuah bab membutuhkan dependensi di luar pustaka standar (mis. JUnit test atau GUI JavaFX di Lecture 11).

Alat	Rekomendasi	Alasan

Anda tidak perlu memilih "satu IDE yang benar"

Setiap listing kode dalam buku ini adalah source .java biasa yang di-compile dengan compiler javac standar — ia akan di-compile dan berjalan identik apa pun IDE (atau tanpa IDE sama sekali, dari terminal) yang Anda pakai untuk mengeditnya. Pilih alat apa pun yang diwajibkan kelas, pemberi kerja, atau lab Anda; tidak ada apa pun di mata kuliah ini yang bergantung pada file proyek khas suatu IDE.

Setelah JDK terpasang, verifikasi dari terminal sebelum membuka IDE apa pun — ini menangkap instalasi yang hilang atau salah konfigurasi sebelum bisa membingungkan Anda di dalam alat yang lebih kompleks:

```
$ java -version
openjdk version "21.0.10" 2026-01-20
OpenJDK Runtime Environment ...

$ javac -version
javac 21.0.10
```

1.7 Contoh Kerja: Class Java Pertama Anda

Contoh kerja mata kuliah ini mengikuti kerangka lima-fase — Understand, Abstract, Design, Analyze, Implement — untuk setiap masalah yang tidak trivial, dimulai sekarang dengan yang paling sederhana: memodelkan satu buku di perpustakaan Pustaka.

Fase 1 — Understand (Pahami)

```
Masalah: pustakawan perlu mencatat, untuk tiap buku, judul, penulis,
dan ISBN-nya, plus apakah buku itu sedang dipinjam, serta mendukung
peminjaman, pengembalian, dan perpanjangan (maks 2x per peminjaman).
```

Fase 2 — Abstract (Abstraksi)

```
Lucuti semua hal spesifik tentang "perpustakaan" dan "buku", lalu
kenali bentuk umumnya: sejumlah DATA yang tidak boleh menjadi tidak
masuk akal secara internal (mis. "sedang dipinjam" tapi "diperpanjang
-3 kali"), plus sekumpulan kecil PERILAKU yang menjadi satu-satunya
cara sah mengubah data tersebut.
```

Fase 3 — Design (Rancang)

```
Field (private, agar hanya method class ini sendiri yang menyentuh):
  title, author, isbn : String   (tidak berubah setelah dibuat)
  onLoan              : boolean  (mulai false)
  timesRenewed        : int      (mulai 0, reset tiap peminjaman baru)
Method (public — SATU-SATUNYA cara kode luar memengaruhi objek ini):
  borrow()      -> boolean   false jika sudah dipinjam
  returnBook()  -> void
  renew()       -> boolean   false jika tak dipinjam atau sudah
                             diperpanjang dua kali
  toString()    -> String    ringkasan mudah-dibaca, untuk dicetak
```

Fase 4 — Analyze (Analisis)

```
Argumen kebenaran: onLoan hanya bisa menjadi true di dalam borrow(),
dan hanya saat sebelumnya false; hanya bisa menjadi false di dalam
returnBook(). timesRenewed hanya bisa bertambah di dalam renew(), dan
maksimal 2, serta di-reset ke 0 setiap kali borrow() berhasil. Karena
setiap field private, TIDAK ADA kode di tempat lain dalam program yang
bisa melanggar aturan ini bahkan secara tidak sengaja -- compiler itu
sendiri menolak kode yang mencobanya.
```

Fase 5 — Implement (Implementasikan)

```
public class Book {
    private final String title;
    private final String author;
    private final String isbn;
    private boolean onLoan;
    private int timesRenewed;

    public Book(String title, String author, String isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.onLoan = false;
        this.timesRenewed = 0;
    }

    public boolean borrow() {
        if (onLoan) { return false; }
        onLoan = true;
        timesRenewed = 0;
        return true;
    }

    public void returnBook() { onLoan = false; }

    public boolean renew() {
        if (!onLoan || timesRenewed >= 2) { return false; }
        timesRenewed++;
        return true;
    }

    @Override
    public String toString() {
        String status = onLoan
            ? "ON LOAN (renewed " + timesRenewed + "x)"
            : "on the shelf";
        return String.format("\"%s\" by %s [ISBN %s] - %s",
            title, author, isbn, status);
    }
}
```

Sebuah class dengan sendirinya tidak melakukan apa-apa sampai sesuatu membuat objek darinya — Java menyebut ini instansiasi, memakai kata kunci new. Program driver berikut (class terpisah yang hanya berisi method main) membuat satu objek Book dan menjalankan perilakunya; ini kutipan yang dipersingkat untuk halaman ini — file lengkapnya, yang juga membuat buku kedua serta mendemonstrasikan renew() dan returnBook(), disertakan sebagai

Code/Java/Chapter01/LibraryDemo.java bersama buku ini, dan keluaran lengkapnya itulah yang diverifikasi Lampiran 1.A:

```
public class LibraryDemo {
    public static void main(String[] args) {
        Book cleanCode = new Book("Clean Code",
            "Robert C. Martin", "978-0132350884");

        System.out.println(cleanCode);
        cleanCode.borrow();
        System.out.println(cleanCode);
        boolean again = cleanCode.borrow();
        System.out.println("Second borrow allowed? " + again);
    }
}
```

Telusuri sendiri sebelum melanjutkan membaca

Apa yang dicetak program di atas pada baris ketiganya? Telusuri: println pertama menampilkan buku yang baru dibuat (di rak); borrow() lalu mengubah onLoan menjadi true; println kedua mencerminkan itu; dan pemanggilan kedua ke borrow() — pada buku yang sudah dipinjam — harus mengembalikan false, karena argumen kebenaran Fase 4 menjamin onLoan tidak bisa menjadi true untuk kedua kalinya tanpa returnBook() di antaranya. Jika penelusuran Anda tidak sesuai dengan "Second borrow allowed? false", baca ulang satu pernyataan if di dalam borrow().

1.8 Meng-compile dan Menjalankan Program Anda

Java adalah bahasa yang di-compile: compiler javac menerjemahkan file source .java Anda menjadi file .class berisi bytecode JVM, dan launcher java kemudian menjalankan bytecode tersebut pada Java Virtual Machine. Kedua file, Book.java dan LibraryDemo.java, harus di-compile sebelum LibraryDemo bisa dijalankan, karena source LibraryDemo merujuk ke tipe Book:

```
$ javac Book.java LibraryDemo.java
$ java LibraryDemo
```

Jika Anda memakai IDE seperti IntelliJ IDEA atau Eclipse, IDE menjalankan kedua perintah yang sama ini untuk Anda di balik tombol "Run" — memahami apa yang sebenarnya terjadi di balik tombol itu akan membuat pesan error IDE jauh lebih tidak misterius sepanjang mata kuliah ini.

1.9 OOP dalam Praktik: Profitina

Tentang kotak ini

Di seluruh buku ini, kotak seperti ini menghubungkan materi mata kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis. Kotak ini menjelaskan DI MANA dan MENGAPA sebuah konsep dipakai dalam sistem nyata yang berjalan — bukan logika bisnis atau source code lengkap Profitina yang proprietary, yang tetap rahasia. Lihat dokumen pendamping "Profitina: A Non-Confidential Technical Overview" untuk gambaran lengkapnya.

Produk Profitina sendiri (app.profitina.com) ditulis dalam Python, bukan Java — tetapi ide OOP yang diperkenalkan bab ini, enkapsulasi, berlaku identik apa pun bahasa yang mengekspresikannya. Backend Profitina mendefinisikan class seperti objek hasil-pindai yang menyatukan skor visibilitas sebuah bisnis bersama method yang diizinkan memperbarui skor tersebut, alih-alih mengekspos skor itu sebagai angka telanjang yang bisa ditimpa bagian mana pun dari basis kode. Model AI yang dijalankan sendiri (self-hosted) oleh Profitina (Qwen2.5 untuk analisis panjang, Qwen3:1.7b untuk klasifikasi cepat, keduanya lewat Ollama, sepenuhnya di server milik Profitina sendiri) sendiri dibungkus di balik class dan method milik Profitina — bagian lain aplikasi tidak pernah menjangkau melewati objek-objek itu untuk mengutak-atik internal model secara langsung. Ini persis disiplin class-Book pada Bagian 1.7, hanya diperbesar skalanya dari contoh pengajaran menjadi produk nyata yang dipakai bisnis sungguhan hari ini.

Anda akan melihat pola yang sama ini di setiap bab berikutnya: ide Java atau Python apa pun yang diperkenalkan sebuah bab, mata kuliah ini akan menunjukkan persis di mana ide setara muncul di dalam basis kode Profitina yang nyata, aktif, dan produksi — selalu pada level "begini bentuk ide itu dipakai," tidak pernah pada level detail implementasi proprietary.

1.10 Ringkasan Bab dan Poin Penting

- Program prosedural yang menyebar data ke berbagai array paralel tidak punya satu tempat pun yang memastikan data tetap masuk akal — fungsi mana pun di mana pun bisa merusaknya.
- Sebuah objek menyatukan data (field) dengan perilaku (method) yang diizinkan mengubah data tersebut; enkapsulasi berarti menyembunyikan field sehingga hanya method objek itu sendiri yang bisa menyentuhnya.
- Empat pilar mata kuliah ini — enkapsulasi, abstraksi, pewarisan, polimorfisme — masing-masing akan mendapat bab tersendiri, dalam urutan itu, karena setiap pilar berikutnya dibangun di atas disiplin yang ditetapkan pilar sebelumnya.
- Java adalah bahasa yang di-compile: javac menghasilkan bytecode .class dari source .java, dan java menjalankan bytecode itu di JVM. Pakai JDK 21 atau 25 (keduanya rilis LTS terkini per 2026), dan IntelliJ IDEA atau Eclipse sebagai IDE Anda.
- Kerangka Lima-Fase (Understand, Abstract, Design, Analyze, Implement) yang dipakai untuk Book di bab ini akan dipakai untuk setiap contoh kerja tidak-trivial di mata kuliah ini.

"Saya yang menciptakan istilah 'object-oriented', dan saya bisa pastikan saya tidak sedang memikirkan C++." — sintaks bahasa yang akan Anda pelajari di mata kuliah ini adalah nomor dua dibanding model mental yang diperkenalkan bab ini: program sebagai objek-objek yang saling berkomunikasi, masing-masing bertanggung jawab melindungi datanya sendiri.

1.11 Soal Latihan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Lecture 2.

1. Dengan kata-kata Anda sendiri, jelaskan mengapa "programmer-nya saja yang perlu hati-hati" tidak dianggap jawaban memuaskan untuk masalah korupsi data yang dijelaskan di Bagian 1.2.

2. Field timesRenewed pada Book bersifat private. Tulis satu kalimat menjelaskan apa yang akan salah pada argumen kebenaran Fase 4 jika field itu bersifat public.
3. Modifikasi (di kertas, atau langsung di IDE Anda) class Book untuk menambahkan method extendDueDate(int days) yang hanya boleh berhasil jika buku sedang dipinjam. Di bagian class mana aturan ini harus ditegakkan, dan mengapa tidak bisa ditegakkan di tempat lain?
4. Gambar 1.3 menunjukkan versi prosedural dengan empat array paralel. Misalkan field baru, dueDate, perlu ditambahkan. Daftarkan setiap tempat pada versi prosedural yang harus berubah, dibandingkan setiap tempat pada versi berorientasi objek yang harus berubah.
5. Pilar mana dari empat pilar (enkapsulasi, abstraksi, pewarisan, polimorfisme) yang didemonstrasikan contoh Book bab ini? Tiga pilar mana yang sengaja BELUM didemonstrasikan, dan mengapa (petunjuk: baca ulang Bagian 1.5)?

Lampiran 1.A — Log Verifikasi Kompilasi dan Eksekusi

File lengkap Book.java dan LibraryDemo.java (Code/Java/Chapter01/, disertakan bersama buku ini — superset dari kutipan yang dipersingkat pada Bagian 1.7, juga mendemonstrasikan buku kedua serta renew() dan returnBook()) di-compile dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kodenya benar, bukan sekadar "terlihat benar". Keluaran direproduksi verbatim di bawah.

```
$ javac Book.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
--- Pustaka: starting state ---
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - on the shelf
"The Pragmatic Programmer" by Hunt & Thomas [ISBN 978-0135957059] - on the shelf

--- A student borrows Clean Code ---
Borrow succeeded? true
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - ON LOAN (renewed 0x)

--- Someone else tries to borrow the SAME copy ---
Borrow succeeded? false (already on loan, so the object refuses)

--- The student renews, then returns it ---
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - ON LOAN (renewed 1x)
"Clean Code" by Robert C. Martin [ISBN 978-0132350884] - on the shelf
```

Referensi

[1] A. Kay. Surel ke milis pengembang Squeak, 2003 (banyak dikutip sebagai "I made up the term object-oriented, and I can tell you I did not have C++ in mind").

[2] Oracle Corporation. "Java SE Support Roadmap." https://www.oracle.com/java/technologies/java-se-support-roadmap.html (diakses Agustus 2026) — mengonfirmasi Java 8, 11, 17, 21, dan 25 sebagai rilis LTS, Java 25 GA September 2025.

[3] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008 (sumber judul contoh yang dipakai bab ini).

[4] JetBrains. "IntelliJ IDEA." https://www.jetbrains.com/idea/ (Community Edition, diakses Agustus 2026).

[5] Eclipse Foundation. "Eclipse IDE for Java Developers." https://www.eclipse.org/downloads/packages/ (diakses Agustus 2026).

BAB 2

Anatomi Sebuah Class: Field, Constructor & Enkapsulasi

Bab 1 memberi tahu Anda bahwa objek menyatukan data dengan perilaku. Bab ini membuka tudungnya: persis bagian mana dari sebuah class yang mendeklarasikan data itu, persis bagian mana yang membangun objek baru, dan persis seberapa banyak data itu yang boleh dilihat dunia luar.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

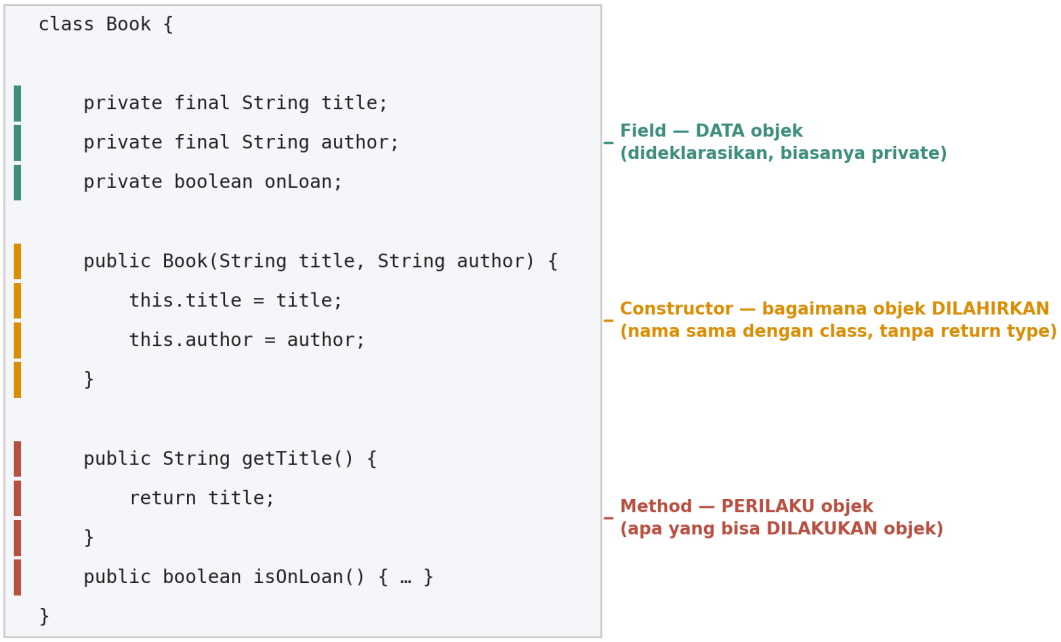
(1) Mengidentifikasi tiga jenis anggota sebuah class — field, constructor, method — dan menyatakan fungsi masing-masing. (2) Menulis constructor, dan menjelaskan apa yang dirujuk "this" di dalamnya. (3) Meng-overload constructor dan memakai this(…) untuk mendelegasikan dari satu constructor ke constructor lain, menghindari duplikasi logika penetapan field. (4) Menulis getter yang mengekspos field objek untuk dibaca tanpa mengizinkan kode luar merusaknya. (5) Menyebutkan empat access modifier Java, menyatakan aturan visibilitas masing-masing, dan menjelaskan apa yang dipakai Python sebagai gantinya. (6) Memperluas class Book dari Bab 1 dengan field baru, constructor tambahan, dan cakupan getter lengkap, lalu meng-compile, menjalankan, dan memverifikasinya sendiri.

2.1 Melanjutkan dari Bab 1

Bab 1 memperkenalkan satu ide tunggal yang menjadi fondasi setiap bab berikutnya: objek menyatukan data (field) dengan perilaku (method) yang diizinkan mengubah data itu, dan menyembunyikan field di balik perilaku tersebut. Anda melihat satu class, Book, dengan tiga field dan empat method, dan Anda meng-compile serta menjalankannya. Bab ini tidak memperkenalkan ide baru sebanyak ia membedah class Book yang sama itu bagian demi bagian dan mengamati dari dekat masing-masing dari tiga jenis anggotanya — field, constructor, dan method — dimulai dari jenis anggota yang terus-menerus dipakai Bab 1 namun tidak pernah dijelaskan: constructor.

Anatomi Sebuah Class: Field, Constructor, dan Method

Setiap class Java atau Python yang Anda tulis punya (paling banyak) tiga jenis anggota ini, dalam urutan konvensional ini.



Gambar 2.1 — Setiap class yang Anda tulis punya (paling banyak) tiga jenis anggota ini, dalam urutan konvensional ini: field, lalu constructor, lalu method.

2.2 Field: Data Objek, Secara Formal

Field adalah variabel yang dideklarasikan langsung di dalam badan class (bukan di dalam method apa pun) — di sanalah data sebuah objek hidup. Book pada Bab 1 mendeklarasikan tiga field: title, author, dan isbn, semuanya bertipe String, plus onLoan (boolean) dan timesRenewed (int). Setiap objek yang dibangun dari class Book mendapat salinan private-nya sendiri dari kelima field ini; dua objek Book yang berbeda tidak pernah berbagi variabel title yang sama, meskipun keduanya dideklarasikan oleh potongan source code tiga baris yang persis sama (Gambar 2.1).

Perhatikan bahwa title, author, dan isbn dideklarasikan private final. private membatasi siapa yang boleh membaca atau menulis field secara langsung (Bagian 2.6 membahasnya secara lengkap); final menambahkan pembatasan kedua yang independen — setelah field final ditetapkan nilainya di dalam constructor, ia tidak akan pernah bisa ditetapkan ulang seumur hidup objek itu. Menandai field sebagai final adalah cara Java membiarkan Anda berjanji, dan membuat compiler menegakkan janji itu, "data ini tidak bisa berubah setelah objek dibangun." onLoan dan timesRenewed sengaja TIDAK dibuat final, karena seluruh tujuan mereka adalah berubah seiring buku dipinjam, diperpanjang, dan dikembalikan.

Python tidak punya kata kunci field dan tidak punya kata kunci final

Atribut Python muncul begitu saja pada saat self._title = title dieksekusi di dalam __init__ — tidak ada sintaks deklarasi-field terpisah untuk dipelajari, dan tidak ada final yang ditegakkan compiler. Jika sebuah class Python ingin menyinyalkan "atribut ini seharusnya tidak ditetapkan ulang setelah konstruksi," ia bersandar pada disiplin konvensi yang sama yang dibahas Bab 1: satu garis bawah di depan, @property tanpa @x.setter yang sepadan, dan disiplin programmer. Kode Python Bab 2 mengikuti konvensi ini secara konsisten.

2.3 Constructor: Bagaimana Objek Dilahirkan

Sebuah class dengan sendirinya hanyalah cetak biru; tidak ada yang eksis sampai kode memanggil new (Java) atau memanggil nama class seperti fungsi (Python), dan pemanggilan itu selalu menjalankan sebuah constructor. Tugas constructor tunggal: mengambil nilai mentah apa pun yang diberikan pemanggil dan memakainya untuk membawa satu objek baru ke keberadaan dalam keadaan valid, sepenuhnya terinisialisasi — tidak pernah objek setengah-jadi dengan beberapa field ditetapkan dan yang lain masih memegang nilai sampah.

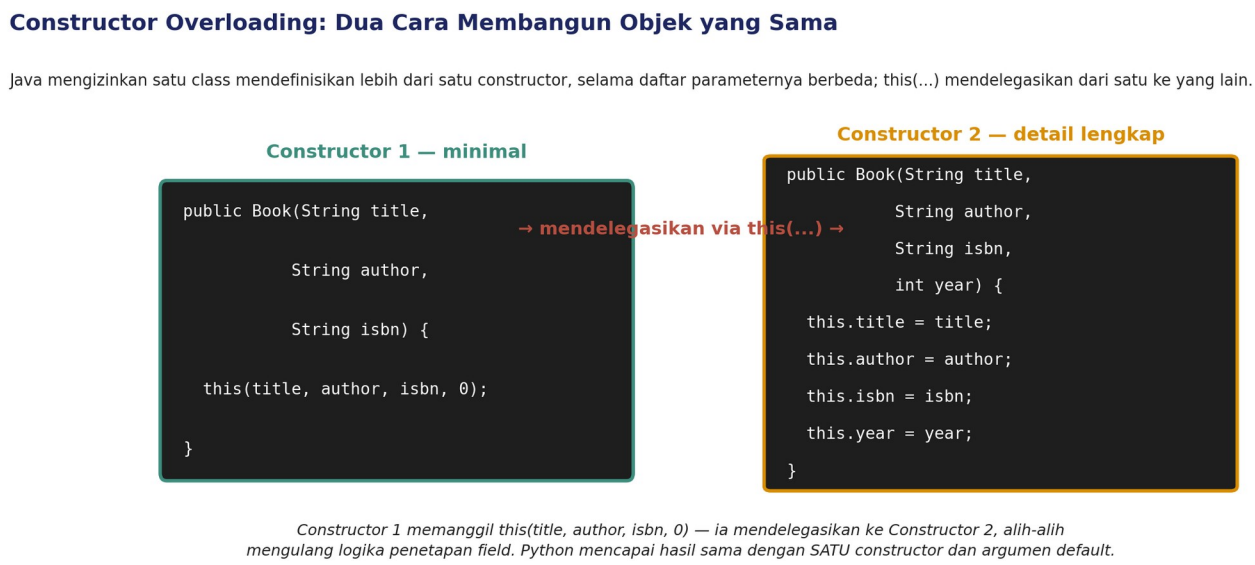
Dalam Java, constructor adalah blok berbentuk-method dengan tiga sifat tidak lazim: ia tidak punya return type (bahkan tidak void), namanya persis nama class-nya, dan badannya berjalan persis satu kali, pada saat new Book(...) dieksekusi. Di dalam constructor, kata kunci this merujuk pada "objek yang sedang dibangun sekarang" — ia ada khusus untuk memecahkan tabrakan penamaan umum di mana parameter constructor (title) dan field yang ingin diinisialisasinya (juga bernama title) berbagi nama yang sama: this.title = title secara tidak ambigu berarti "field milik objek ini mendapat nilai dari parameter."

Lupa this. adalah bug klasik pertama

Jika parameter constructor dan field berbagi nama dan Anda menulis title = title; alih-alih this.title = title;, Java me-resolve nama telanjang title ke deklarasi terdekat dalam scope — parameternya — sehingga pernyataan itu menetapkan parameter kepada dirinya sendiri dan field-nya sama sekali tidak tertetapkan (null, untuk field String). Ini di-compile tanpa error dan gagal secara diam-diam, yang justru sebabnya layak ditelusuri dengan cermat kali pertama Anda melihatnya. Python menghindari jebakan spesifik ini karena self.title dan title tidak pernah ambigu — self. wajib dipakai untuk menjangkau atribut, sehingga tidak ada nama telanjang yang bisa tak sengaja tertimpa.

2.4 Constructor Overloading dan Chaining this(...)

Pemanggil nyata tidak selalu punya informasi yang sama tersedia. Terkadang Anda tahu tahun terbit sebuah buku; terkadang tahunnya sekadar tidak tercatat di mana pun yang bisa Anda akses. Java mengizinkan satu class mendefinisikan beberapa constructor selama daftar parameternya bisa dibedakan (jumlah atau tipe parameter berbeda) — ini disebut constructor overloading, dan compiler memilih yang tepat untuk dijalankan berdasarkan berapa banyak argumen, dan tipe apa, yang Anda berikan ke new Book(...) (Gambar 2.2).



Gambar 2.2 — Constructor 1 (tiga argumen) mendelegasikan ke Constructor 2 (empat argumen) via this(title, author, isbn, 0), sehingga logika penetapan field ditulis persis satu kali.

Badan satu baris this(title, author, isbn, 0); di dalam constructor tiga-argumen adalah pemanggilan constructor, bukan penetapan field — ia harus menjadi pernyataan pertama dalam badan constructor, dan ia menyerahkan pekerjaannya ke constructor empat-argumen, memberikan 0 sebagai nilai pengganti yang berarti "tahun terbit tidak tercatat." Tanpa chaining this(...), kedua constructor perlu mengulang kelima baris penetapan field, dan perbaikan bug di masa depan pada satu salinan bisa dengan mudah terlupa di salinan lainnya.

> **Constructor overloading, ala Python: satu constructor, argumen default**
>
> Signature method Python tidak mendukung overloading sejati seperti Java — hanya ada tepat satu __init__ per class. Python sebagai gantinya mencapai hasil yang persis sama dengan satu constructor dan nilai parameter default: def __init__(self, title, author, isbn, publication_year=0). Memanggil Book(t, a, i) dan Book(t, a, i, 2008) keduanya bekerja, dan keduanya menjalankan badan __init__ yang sama — Python tidak punya constructor 'minimal' terpisah yang perlu disinkronkan dengan yang 'lengkap', sehingga tidak ada chaining ala this(...) untuk dipelajari sejak awal.

2.5 Getter: Akses Baca Terkendali

Menandai field sebagai private (Java) atau garis-bawah-di-depan (Python) memecahkan sisi-tulis enkapsulasi — tidak ada apa pun di luar class yang bisa merusak field dengan menetapkan nilai yang tidak masuk akal. Tapi ini juga, sebagai efek samping yang tak terhindarkan, memblokir dunia luar untuk sekadar membaca nilai field saat ini, yang sangat sering menjadi sesuatu yang secara sah dibutuhkan kode luar (halaman katalog perlu menampilkan judul buku; fitur pencarian perlu memeriksa penulisnya). Getter adalah method public kecil yang seluruh tugasnya adalah mengembalikan nilai satu field saat ini, memulihkan akses baca tanpa membuka kembali akses tulis.

Getter sengaja dibuat sesederhana mungkin: ia tidak menerima parameter, dan badannya adalah satu pernyataan return. Konvensi penamaan Java memberi prefiks get pada nama field (getTitle(), getAuthor()), kecuali untuk field boolean, yang secara konvensional memakai is alih-alih get (isOnLoan(), bukan getOnLoan()) — ini adalah konvensi komunitas, bukan aturan bahasa, tapi hampir

setiap basis kode Java yang akan Anda baca mengikutinya, dan melanggarnya tanpa alasan membingungkan setiap programmer Java lain yang membaca kode Anda.

Getter Java	@property Python	Mengembalikan
getTitle()	title	judul buku (String / str)
getAuthor()	author	penulis buku (String / str)
getIsbn()	isbn	ISBN buku (String / str)
getPublicationYear()	publication_year	tahunnya, atau 0 jika tak tercatat (int)
isOnLoan()	is_on_loan	apakah buku sedang dipinjam (boolean / bool)
getTimesRenewed()	times_renewed	berapa kali peminjaman ini diperpanjang (int)

Getter yang mengembalikan String atau int tetap aman, bahkan tanpa "menyalinnya"

Kekhawatiran wajar: jika getTitle() sekadar mengembalikan field-nya langsung, bukankah pemanggil yang menerimanya bisa balik merusak state internal Book lewat referensi yang dikembalikan itu? Untuk String, int, dan boolean secara spesifik, tidak — tipe-tipe ini immutable baik di Java maupun Python, artinya tidak ada method pada String atau int yang bisa mengubah karakter atau digit yang sudah dipegangnya. Pemanggil menerima nilai yang bisa mereka baca, salin, atau buang, tapi tidak pernah sebuah pegangan yang membiarkan mereka menjangkau balik ke dalam Book dan mengubah field private-nya. (Pertanyaan ini menjadi jauh lebih menarik begitu sebuah getter mengembalikan tipe mutable, seperti list peminjaman — masalah yang dibahas langsung mata kuliah ini di Bab 10, Generics & Collections.)

2.6 Access Modifier: Empat Level Java vs. Konvensi Python

Bagian 2.2 dan Bab 1 sama-sama sudah memakai kata private berulang kali tanpa mengkatalogkan alternatifnya secara lengkap. Java mendefinisikan empat level visibilitas yang bisa diberikan pada field, constructor, atau method, dan getter Bagian 2.5 ada khusus untuk membuka lubang sempit dan terkendali menembus yang paling ketat dari keempatnya (Gambar 2.3).

Access Modifier Sekilas: Empat Level Java vs. Konvensi Python

Compiler Java menegakkan visibilitas; Python menyinyalkan maksud yang sama lewat konvensi penamaan dan mempercayai programmer.

Modifier Java	Terlihat dari	Padanan Python
`private`	Hanya class itu sendiri	`_satu_garis_bawah` (konvensi: internal)
`(package-private)`	Hanya package yang sama	— (Python tidak punya kontrol akses tingkat package)
`protected`	Package + subclass	— (jarang dibutuhkan sampai Bab 9, pewarisan)
`public`	Di mana pun	tanpa garis bawah (default)

private milik Java ditegakkan compiler — kode di luar class bahkan tidak bisa mencoba meng-compile pelanggarannya. _garis_bawah_di_depan Python adalah janji antar-programmer, bukan tembok yang dibangun interpreter untuk Anda.

Gambar 2.3 — Empat level akses Java, dari yang paling sempit (private) hingga paling luas (public), berdampingan dengan padanan konvensi-penamaan Python untuk masing-masing.

private adalah level yang dipakai mata kuliah ini untuk setiap field sejauh ini, dan ia yang paling ketat: hanya terlihat di dalam badan class itu sendiri. package-private (default Java, saat tidak ada kata kunci sama sekali yang ditulis) membuka visibilitas ke setiap class lain dalam package yang sama — jalan tengah yang belum disandari mata kuliah ini sampai bab-bab berikutnya mengelompokkan class-class terkait. protected memperluas visibilitas package-private ke subclass juga, di mana pun mereka berada — level yang kegunaan penuhnya baru terlihat begitu Bab 9 memperkenalkan pewarisan. public, level terluas, adalah yang dipakai setiap getter dan setiap method perilaku (borrow(), returnBook(), renew()) di mata kuliah ini, karena method-method itu persis pintu terkendali yang dimaksudkan untuk dipakai bagian lain program.

> **public untuk semuanya mengalahkan seluruh inti Bab 1**
>
> Secara teknis legal menandai setiap field sebagai public dan menghapus setiap getter serta setiap method pembatas-akses — Java akan meng-compile ini tanpa keluhan. Melakukannya membuang semua yang diperjuangkan Bab 1: dengan setiap field public, kode di mana pun dalam program bisa menetapkan onLoan = true secara langsung, sepenuhnya melewati pemeriksaan borrow() tentang apakah buku sudah dipinjam, memunculkan kembali persis risiko korupsi-data yang dibangun perbandingan prosedural-vs-OOP Bab 1 (Gambar 1.3) untuk didemonstrasikan. Enkapsulasi bukan kata kunci private itu sendiri; ia adalah field private yang sengaja dipasangkan dengan sekumpulan kecil method public yang terkurasi.

2.7 Contoh Kerja: Memperluas Book dengan Tahun Terbit

Contoh kerja bab ini mengambil class Book dari Bab 1 dan memperluasnya: field publicationYear baru, dua constructor (mendemonstrasikan overloading dan chaining this(...)), dan getter untuk setiap field.

Fase 1 — Understand (Pahami)

```
Kebutuhan baru: sebagian program (fitur pencarian katalog di masa
depan) perlu akses baca ke judul, penulis, isbn, status pinjam, dan
jumlah perpanjangan sebuah buku -- dan perlu mencatat tahun terbit
buku bila diketahui, tanpa mewajibkannya bila tidak diketahui.
```

Fase 2 — Abstract (Abstraksi)

```
Bentuk umum: satu DATA baru (publicationYear), plus kanal BACA
terkendali (getter) untuk setiap data yang sudah ada, plus cara
membangun objek entah data baru itu tersedia atau tidak saat
konstruksi.
```

Fase 3 — Design (Rancang)

```
Field baru:
  publicationYear : int (final; 0 berarti 'tidak tercatat')
Constructor (overload, yang 3-argumen mendelegasikan via this(...)):
  Book(title, author, isbn)               -- tahun default 0
  Book(title, author, isbn, publicationYear)
Getter (public; SATU-SATUNYA kanal baca untuk field private):
  getTitle(), getAuthor(), getIsbn(), getPublicationYear(),
  isOnLoan(), getTimesRenewed()   -- semua tanpa argumen, satu return
```

Fase 4 — Analyze (Analisis)

```
Argumen kebenaran: publicationYear bersifat final, jadi setelah
Book dikonstruksi ia tidak bisa diam-diam berubah, persis seperti
title, author, dan isbn. Setiap getter mengembalikan tipe immutable
(String, int, atau boolean), jadi tidak ada pemanggil yang menerima
nilai kembalian getter yang bisa menjangkau balik dan mengubah state
private Book lewatnya. Kedua constructor tidak bisa tidak-sinkron,
karena yang 3-argumen mendelegasikan seluruh badannya ke yang
4-argumen via this(...) -- hanya ada satu tempat kelima field
benar-benar ditetapkan.
```

Fase 5 — Implement (Implementasikan)

```
public class Book {
    private final String title;
    private final String author;
    private final String isbn;
    private final int publicationYear;
    private boolean onLoan;
    private int timesRenewed;

    public Book(String title, String author, String isbn) {
        this(title, author, isbn, 0);
    }

    public Book(String title, String author, String isbn,
                int publicationYear) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
        this.publicationYear = publicationYear;
        this.onLoan = false;
        this.timesRenewed = 0;
    }

    // borrow(), returnBook(), renew() -- tak berubah dari Bab 1

    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public String getIsbn() { return isbn; }
    public int getPublicationYear() { return publicationYear; }
    public boolean isOnLoan() { return onLoan; }
    public int getTimesRenewed() { return timesRenewed; }
}
```

Program driver di bawah membuat dua objek Book, satu memakai constructor empat-argumen (tahun terbit diketahui) dan satu memakai constructor tiga-argumen (tahun tak tercatat, default ke 0 via chaining this(…)), lalu menjalankan beberapa getter. Ia ditampilkan di sini secara lengkap, persis seperti tampil di Code/Java/Chapter02/LibraryDemo.java, dan keluaran lengkapnya persis yang diverifikasi Lampiran 2.A:

```
public class LibraryDemo {
    public static void main(String[] args) {
        Book cleanCode = new Book("Clean Code", "Robert C. Martin",
            "978-0132350884", 2008);
        Book pragmatic = new Book("The Pragmatic Programmer",
            "Hunt & Thomas", "978-0135957059");

        System.out.println(cleanCode);
        System.out.println(pragmatic);

        System.out.println("Clean Code author: "
            + cleanCode.getAuthor());
        System.out.println("Pragmatic Programmer year: "
            + pragmatic.getPublicationYear() + " (0 = unrecorded)");

        cleanCode.borrow();
        System.out.println(cleanCode);
    }
}
```

Telusuri sendiri sebelum melanjutkan membaca

pragmatic dibangun dengan constructor tiga-argumen, yang sama sekali tidak menyebut tahun terbit. Apa yang dikembalikan pragmatic.getPublicationYear(), dan apa yang dicetak baris toString() untuk tahun pragmatic, mengingat itu? Telusuri delegasinya: constructor 3-argumen memanggil this(title, author, isbn, 0) — jadi 0 persis nilai yang tersimpan di publicationYear, dan toString() (Fase 5 Bagian 2.7, diperluas lebih lanjut di file source lengkap) ditulis untuk menghilangkan sepenuhnya kurung tahun setiap kali publicationYear bernilai 0, alih-alih mencetak angka 0 yang menyesatkan.

2.8 Meng-compile dan Menjalankan Program Anda

Tidak ada yang berubah dari proses compile-dan-jalankan yang diperkenalkan Bab 1 Bagian 1.8: javac Book.java LibraryDemo.java meng-compile kedua file (masih harus bersama, karena source LibraryDemo merujuk ke tipe Book), dan java LibraryDemo menjalankan bytecode yang sudah di-compile.

```
$ javac Book.java LibraryDemo.java
$ java LibraryDemo
```

Jika Anda memperluas Book.java sendiri sambil mengerjakan Soal Latihan bab ini, biasakan meng-compile ulang KEDUA file bersama dengan satu pemanggilan javac, persis seperti ditunjukkan di atas, alih-alih meng-compile hanya file yang Anda edit — Java akan menolak menjalankan LibraryDemo terhadap Book.class yang basi, tertinggal dari sebelum edit Anda, dan meng-compile ulang keduanya bersama setiap kali menghilangkan segala kemungkinan kebingungan itu.

2.9 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti di setiap bab, kotak ini menghubungkan materi mata kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis — menjelaskan DI MANA dan MENGAPA sebuah konsep dipakai dalam sistem nyata yang berjalan, bukan logika bisnis atau source code lengkap Profitina yang proprietary, yang tetap rahasia. Lihat dokumen pendamping "Profitina: A Non-Confidential Technical Overview" untuk gambaran lengkapnya.

Backend Profitina mendefinisikan class yang constructor-nya mencerminkan persis pola yang diajarkan bab ini: sebuah class hasil-pindai, misalnya, dibangun sekali per pemindaian AI-visibility yang selesai, dengan setiap field yang dibutuhkannya (skor bisnis, timestamp pemindaian, model AI mana yang dikonsultasikan) diberikan saat konstruksi lalu diperlakukan sebagai tetap secara efektif seumur hidup objek itu — disiplin yang sama yang didemonstrasikan field publicationYear final bab ini dalam skala kecil. Ketika kode berikutnya perlu membaca skor hasil-pindai untuk ditampilkan di dashboard, ia melakukannya lewat accessor setara-getter, tidak pernah dengan menjangkau langsung ke field internal objek, persis alasan yang dijelaskan Bagian 2.5: kanal baca terkendali yang tidak pernah bisa menjadi kanal tulis tak-terkendali.

Ini juga tempat kode Profitina sendiri bersandar pada disiplin access-modifier setara-Java meskipun backend-nya ditulis dalam Python: field yang dimaksudkan tetap internal memakai konvensi garis-bawah-di-depan Python secara konsisten (kotak callout Bagian 2.2), dan setiap class Profitina sendiri mengekspos datanya secara eksklusif lewat getter @property, bukan akses atribut telanjang — konvensi yang sama yang diikuti kode Python bab ini untuk Book.

2.10 Ringkasan Bab dan Poin Penting

- Sebuah class punya (paling banyak) tiga jenis anggota: field (data objek), constructor (bagaimana objek dibangun), dan method (perilaku objek).
- Tugas constructor adalah membawa satu objek ke keberadaan dalam keadaan valid; this. (Java) memecahkan tabrakan penamaan umum antara parameter constructor dan field yang diinisialisasinya.
- Constructor overloading (Java) membiarkan satu class menawarkan beberapa cara membangun objek; chaining this(...) membiarkan satu constructor mendelegasikan ke constructor lain sehingga logika penetapan field ditulis persis satu kali. Python mencapai hasil yang sama dengan satu constructor dan nilai argumen default.
- Getter adalah method public kecil yang mengembalikan nilai satu field, memulihkan akses baca tanpa membuka kembali akses tulis — field-nya tetap private; hanya sebuah pintu sempit yang terkurasi yang dibuka.
- Java punya empat level akses — private, package-private, protected, public — dengan visibilitas yang makin luas; Python menyinyalkan maksud yang sama lewat konvensi penamaan (garis bawah di depan) alih-alih penegakan compiler.
- Menandai setiap field sebagai public mengalahkan tujuan enkapsulasi yang ditetapkan Bab 1 — field private yang sengaja dipasangkan dengan sekumpulan kecil method public itulah yang membuat kebenaran sebuah objek bisa diperdebatkan sama sekali.

Sebuah class adalah janji tentang apa yang boleh terjadi pada datanya, dan constructor adalah tempat janji itu mulai ditepati. Semua yang ditambahkan bab ini pada Book — field baru, dua constructor, enam getter — ada untuk menepati persis janji yang sama yang dibuat Bab 1, hanya untuk objek yang sedikit lebih kaya.

2.11 Soal Latihan

Soal-soal ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Lecture 3.

1. Kedua constructor Book pada akhirnya sama-sama menetapkan kelima field, tapi hanya satu di antaranya yang berisi pernyataan penetapan sebenarnya. Jelaskan, dengan kata-kata Anda sendiri, mengapa itu pilihan desain yang disengaja, bukan kelalaian.
2. Apa yang akan terjadi — sebutkan secara spesifik baris mana yang gagal dan mengapa — jika Anda mencoba menulis publicationYear = publicationYear; setelah pemanggilan this(...) di dalam constructor tiga-argumen Book, mengingat publicationYear dideklarasikan final?
3. isOnLoan() dinamai dengan prefiks is- alih-alih get-. Tulis satu kalimat menjelaskan konvensi apa yang disinyalkan ini, dan sebutkan satu method lain yang mengembalikan boolean di class Book mata kuliah ini yang seharusnya mengikuti konvensi yang sama (petunjuk: cek ulang Bab 1).
4. Misalkan sebuah method getPublisher() ditambahkan yang mengembalikan java.util.List<String> penerbit-penerbit sebelumnya (mutable) alih-alih satu String. Jelaskan mengapa argumen keamanan kotak INSIGHT Bagian 2.5 tidak lagi sepenuhnya berlaku, dan apa yang bisa salah.
5. Tulis method getRemainingRenewals() untuk Book yang mengembalikan berapa banyak perpanjangan yang masih tersedia (maksimum 2, dikurangi timesRenewed). Haruskah method baru ini public atau private, dan apakah menambahkannya membutuhkan penyentuhan access modifier field mana pun yang sudah ada? Justifikasi jawaban Anda.

Lampiran 2.A — Log Verifikasi Kompilasi dan Eksekusi

File lengkap Book.java dan LibraryDemo.java (Code/Java/Chapter02/, disertakan bersama buku ini) di-compile dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kodenya benar, bukan sekadar "terlihat benar". Keluaran direproduksi verbatim di bawah.

```
$ javac Book.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
"Clean Code" by Robert C. Martin [ISBN 978-0132350884, 2008] - on the shelf
"The Pragmatic Programmer" by Hunt & Thomas [ISBN 978-0135957059] - on the shelf
Clean Code author: Robert C. Martin
Pragmatic Programmer year: 0 (0 = unrecorded)
"Clean Code" by Robert C. Martin [ISBN 978-0132350884, 2008] - ON LOAN (renewed
0x)
```

Referensi

[1] Oracle Corporation. "The Java Tutorials — Classes and Objects." https://docs.oracle.com/javase/tutorial/java/javaOO/ (diakses Agustus 2026).

[2] Oracle Corporation. "Java SE Support Roadmap." https://www.oracle.com/java/technologies/java-se-support-roadmap.html (diakses Agustus 2026) — mengonfirmasi Java 8, 11, 17, 21, dan 25 sebagai rilis LTS, Java 25 GA September 2025.

[3] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008 (sumber judul contoh yang dipakai bab ini).

[4] JetBrains. "IntelliJ IDEA." https://www.jetbrains.com/idea/ (Community Edition, diakses Agustus 2026).

[5] Eclipse Foundation. "Eclipse IDE for Java Developers." https://www.eclipse.org/downloads/packages/ (diakses Agustus 2026).

BAB 3

Tipe Primitif, Alur Kontrol, String & Alat Debugging

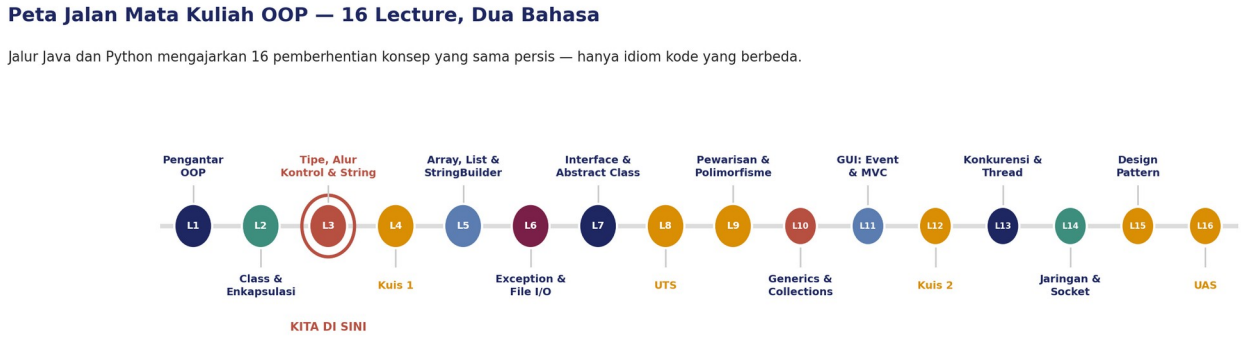
Bab 1 dan 2 membangun dan memperluas satu class. Bab ini mundur selangkah ke fondasi bahasa yang sebenarnya dijalankan oleh setiap badan method: delapan tipe primitif Java, dua cara bercabang, tiga cara berulang, mengapa String berperilaku seperti itu, dan bagaimana mengetahui apa yang sebenarnya dilakukan program Anda ketika hasilnya tidak sesuai harapan.

Tujuan Pembelajaran — setelah bab ini Anda mampu:

(1) Menyebutkan delapan tipe primitif Java serta ukuran dan nilai default masing-masing. (2) Menulis rantai if/else if/else dan kedua bentuk switch (bentuk-titik-dua klasik dan bentuk ekspresi modern Java 14+) untuk menyandikan logika percabangan. (3) Memilih dengan tepat di antara loop for, while, dan do-while, serta menjelaskan hubungan enhanced for-loop dengan for loop klasik. (4) Menjelaskan imutabilitas String dan string pool, serta memilih dengan benar antara == dan .equals() saat membandingkan String. (5) Menggunakan print debugging, breakpoint IDE, dan pernyataan assert untuk mendiagnosis program yang salah. (6) Memperluas class Book milik Pustaka dengan method perhitungan denda dan method status-perpanjangan, lalu meng-compile, menjalankan, dan memverifikasinya sendiri.

3.1 Melanjutkan dari Lecture 2

Bab 2 membuka class Book dan mengamati tiga jenis anggotanya: field, constructor, dan method. Setiap method yang Anda tulis di Bab 1 dan 2 -- borrow(), renew(), getter-getter -- punya badan, dan setiap badan itu dibangun dari kumpulan kecil bahan yang persis sama: nilai-nilai dari segelintir tipe primitif, keputusan (jika ini, maka itu), pengulangan (lakukan ini beberapa kali), dan teks (String). Bab ini mengamati langsung bahan-bahan tersebut, memakai Book sendiri sebagai contoh berjalan di mana pun membantu, dan ditutup dengan keterampilan praktis yang telah diasumsikan dimiliki oleh setiap bab sebelumnya: bagaimana mengetahui mengapa program melakukan sesuatu selain yang Anda harapkan.



Gambar 3.1 — Peta jalan 16-lecture OOP. Bab ini adalah Lecture 3, pemberhentian terakhir sebelum Kuis 1 (Lecture 4) menguji Lecture 1 hingga 3 sekaligus.

3.2 Delapan Tipe Primitif Java

Setiap nilai di Java adalah salah satu dari referensi objek (seluruh subjek Bab 1-2 sejauh ini) atau salah satu dari tepat delapan tipe primitif. Primitif bukan objek: ia tidak punya method, tidak punya field miliknya sendiri, dan variabel bertipe primitif menyimpan nilai sesungguhnya secara langsung, bukan referensi ke objek di tempat lain dalam memori. Book sudah memakai tiga dari delapan tipe ini: int (publicationYear, timesRenewed) dan boolean (onLoan) (Gambar 3.1 dan 3.2).

Delapan Tipe Primitif Java, Sekilas

Setiap nilai yang bukan referensi objek adalah salah satu dari tepat delapan tipe ini.

Tipe	Ukuran	Rentang / Nilai	Default
byte	1 byte	-128 hingga 127	0
short	2 byte	-32.768 hingga 32.767	0
int	4 byte	~ -2,1 miliar hingga 2,1 miliar	0
long	8 byte	~ ±9,2 × 10^18	0L
float	4 byte	~7 digit desimal signifikan	0.0f
double	8 byte	~15 digit desimal signifikan	0.0
char	2 byte	satu unit kode UTF-16	'\u0000'
boolean	1 bit (tergantung JVM)	true atau false	false

int dan double adalah dua kuda beban -- pakai byte/short/float hanya bila ada kendala ukuran atau memori tertentu, dan char hanya untuk satu karakter tunggal, jangan pernah untuk teks umum (pakai String untuk itu).

Gambar 3.2 — Delapan tipe primitif Java, dengan ukuran, rentang, dan nilai default masing-masing. int dan double, yang disorot, adalah dua yang akan paling sering Anda pakai.

Dua aturan praktis mencakup mayoritas besar kode sungguhan. Pertama, pakai int sebagai default untuk bilangan bulat dan double untuk bilangan berpecahan; pakai byte, short, atau float hanya bila ada kendala memori atau presisi tertentu, yang jarang terjadi pada mata kuliah pengantar. Kedua, char menyimpan tepat satu unit kode UTF-16 dan bukan pengganti teks secara umum -- judul buku adalah String (Bab 1), bukan pernah char.

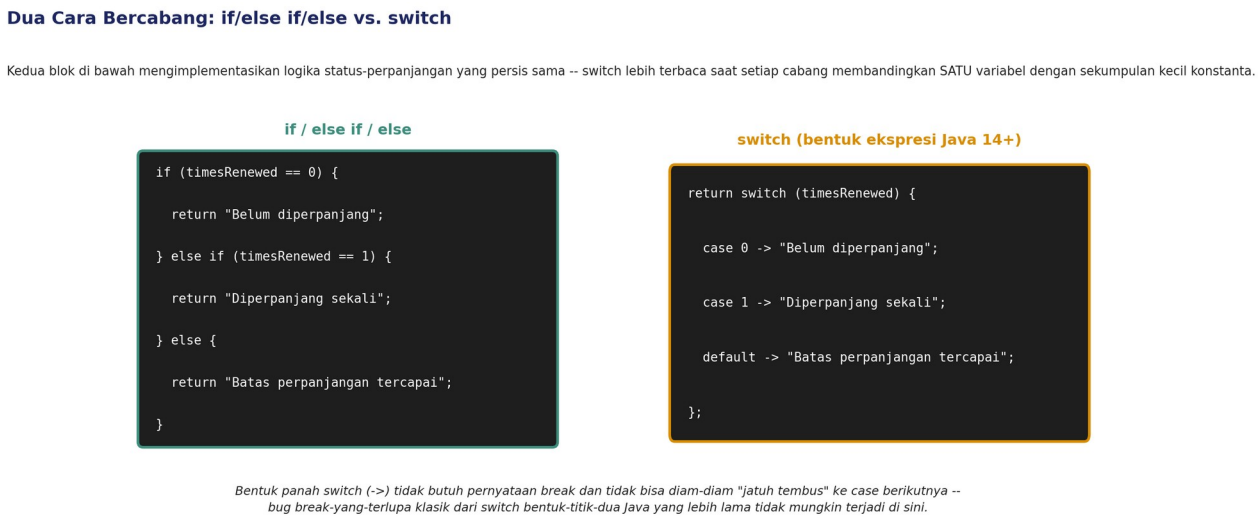
Membandingkan float atau double dengan == adalah bug diam-diam klasik

0.1 + 0.2 == 0.3 bernilai false di Java (dan di hampir semua bahasa lain yang memakai representasi floating-point IEEE 754 yang sama), karena 0.1 dan 0.2 tidak bisa direpresentasikan tepat dalam biner. Jangan pernah membandingkan nilai float atau double untuk kesamaan tepat; bandingkan Math.abs(a - b) terhadap toleransi kecil sebagai gantinya. Method calculateFine bab ini sengaja mengembalikan double untuk jumlah moneter justru untuk mendemonstrasikan pola ini secara bertanggung jawab -- Bagian 3.7 menampilkan jumlah denda yang dicetak dan dibandingkan lewat pemeriksaan visual, tidak pernah diperiksa dengan ==.

3.3 Alur Kontrol: if/else if/else dan switch

Kondisi pernyataan if harus berupa ekspresi boolean -- tidak seperti beberapa bahasa lain, Java tidak pernah diam-diam memperlakukan int atau String sebagai true atau false. Merangkai else if membiarkan sebuah keputusan mengalir lewat beberapa kemungkinan yang saling eksklusif, diakhiri else opsional yang menangkap semua yang belum tertangkap cabang sebelumnya.

switch menawarkan cara kedua untuk membuat jenis keputusan yang sama ketika setiap cabang membandingkan satu variabel terhadap sekumpulan kecil nilai konstan. switch asli Java (bentuk titik-dua, case X: ... break;) terkenal karena satu bug spesifik: lupa break membiarkan eksekusi diam-diam "jatuh tembus" ke case berikutnya. Java 14 memperkenalkan bentuk ekspresi switch (case X -> nilai;) yang tidak punya break untuk dilupakan dan tidak bisa jatuh tembus -- setiap lengan adalah satu ekspresi tunggal, dan switch itu sendiri menjadi nilai yang bisa langsung dikembalikan atau ditetapkan (Gambar 3.3).



Gambar 3.3 — Keputusan status-perpanjangan yang sama, ditulis sebagai if/else if/else dan sebagai ekspresi switch Java 14+. Keduanya benar; switch mengomunikasikan "satu variabel, beberapa case konstan" secara lebih langsung.

> **Kapan memilih switch dibanding if/else if/else**
>
> Pakai switch ketika setiap cabang menguji variabel tunggal yang sama terhadap nilai konstan tertentu (int, String, atau enum) -- persis situasi renewalStatusLabel() di Bagian 3.7. Pilih if/else if/else ketika cabang menguji variabel berbeda, atau menguji rentang/ketidaksetaraan alih-alih kecocokan-tepat konstan -- persis situasi calculateFine(), yang membandingkan daysOverdue terhadap ambang batas (<= 7, <= 30), bukan nilai tepat yang tidak bisa langsung diekspresikan switch.

3.4 Loop: for, while, dan do-while

Java menawarkan tiga bentuk loop, dan Bab 1 sudah memakai bentuk keempat (enhanced for-loop, for (Type x : collection)) tanpa menyebutnya. for loop klasik -- for (inisialisasi; kondisi; update) -- adalah pilihan tepat kapan pun Anda tahu di muka berapa kali Anda bermaksud beriterasi, atau butuh indeks: for (int i = 0; i < 5; i++) menjalankan badannya tepat lima kali, dengan i bernilai 0 hingga 4.

while (kondisi) { ... } memeriksa kondisinya sebelum setiap iterasi, termasuk yang pertama -- ia bisa berjalan nol kali bila kondisi sejak awal false. do { ... } while (kondisi); memeriksa kondisinya setelah setiap iterasi, sehingga badannya selalu berjalan minimal sekali, berguna untuk pola tanya-lalu-validasi ("minta input pengguna, dan terus minta sampai valid") di mana Anda perlu bertanya sebelum punya sesuatu untuk diperiksa.

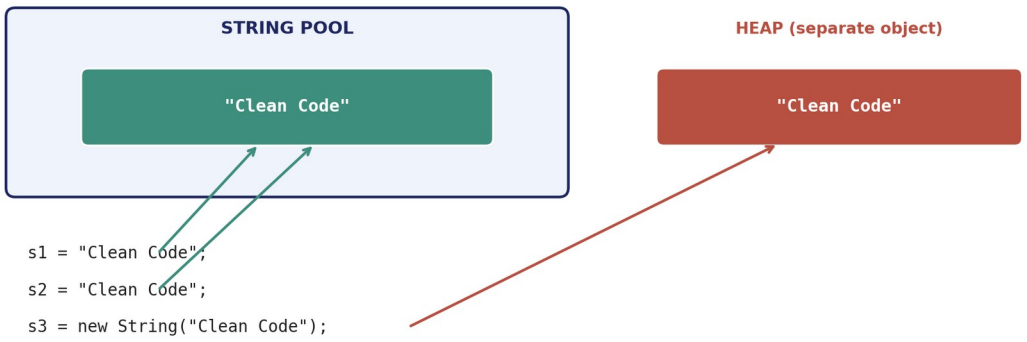
> **Kesalahan off-by-one adalah bug loop paling umum**
>
> for (int i = 0; i <= 5; i++) berjalan ENAM kali (0 hingga 5 inklusif), bukan lima -- perbedaan satu karakter (<= alih-alih <) yang diam-diam menghasilkan satu iterasi ekstra. Ketika jumlah iterasi sebuah loop terlihat salah, memeriksa kondisi batas yang tepat (< vs <=, dan apakah indeks awal 0 atau 1) adalah hal PERTAMA yang harus diperiksa, sebelum mencurigai sesuatu yang lebih eksotis.

3.5 String: Imutabilitas dan Perbandingan

String Java, begitu dibuat, tidak pernah bisa diubah -- setiap method String yang tampak mengubah String (toUpperCase(), trim(), replace(...)) sebenarnya mengembalikan String baru sama sekali, membiarkan yang asli tidak tersentuh. Ini adalah imutabilitas String, dan ia punya konsekuensi langsung yang terlihat: untuk menghemat memori, Java menjaga string pool untuk literal String, dan dua literal String dengan teks identik merujuk ke objek pool yang persis sama, bukan dua objek terpisah.

Imutabilitas String: Pool String, Divisualisasikan

Literal string di-intern dalam pool bersama; new String(...) sengaja membuat objek terpisah di luar pool itu.



s1 == s2 bernilai true (objek pool yang sama); s1 == s3 bernilai false (objek berbeda, konten identik);
s1.equals(s3) bernilai true dalam kedua kasus -- .equals() membandingkan KONTEN, == membandingkan IDENTITAS ("apakah ini objek yang persis sama?").

Gambar 3.4 — s1 dan s2, keduanya literal String dengan teks sama, merujuk ke objek pool yang SAMA. s3, dibangun dengan new String(...), sengaja merujuk ke objek terpisah dengan konten identik.

Inilah persisnya mengapa Java punya dua operasi perbandingan berbeda yang keduanya bisa tampak benar sekilas. == membandingkan referensi -- "apakah kedua variabel ini menunjuk ke objek yang persis sama?" -- sementara .equals() membandingkan konten -- "apakah kedua objek ini menyimpan nilai yang sama?" Khusus untuk String, selalu pakai .equals() (atau Objects.equals(a, b) bila salah satunya mungkin null) untuk membandingkan konten; == pada String di-compile dan berjalan tanpa error tapi diam-diam membandingkan identitas, yang kebetulan cocok untuk literal ber-pool dan sama

seringnya tidak cocok untuk String yang dibangun dengan cara lain (dari input pengguna, dari isi file, dari new String(...)).

Bug input Scanner: if (userInput == "exit") hampir tidak pernah berfungsi

Teks yang dibaca lewat Scanner.nextLine() (atau sumber input sungguhan mana pun) tidak pernah diambil dari string pool -- ia selalu objek String yang baru dibangun, bahkan ketika kontennya persis cocok dengan literal di tempat lain dalam kode Anda. if (userInput == "exit") karena itu membandingkan dua objek berbeda dan bernilai false bahkan ketika pengguna mengetik persis "exit" -- salah satu bug Java awal paling umum, dan salah satu yang dijelaskan sepenuhnya oleh perbedaan bagian ini, bukan sekadar aturan untuk dihafal.

3.6 Alat Debugging: Mengetahui Apa yang Sebenarnya Dilakukan Program Anda

Setiap program yang pernah Anda tulis sejauh ini, pada suatu titik, pernah melakukan sesuatu selain yang Anda harapkan. Tiga alat mencakup hampir semua situasi yang ditemui mata kuliah pengantar, dan mengetahui alat mana yang cocok untuk momen tertentu adalah keterampilan tersendiri yang layak dibangun secara sengaja.

Alat	Cocok untuk	Catatan
Debugging System.out.println	Pemeriksaan cepat dan langsung satu-dua nilai, di mana pun dalam kode	Harus dihapus (atau diubah menjadi logging sungguhan) sebelum kode dianggap selesai
Breakpoint IDE (step-over, step-into, watch)	Mengikuti jalur eksekusi program yang tepat dan memeriksa banyak variabel sekaligus, tanpa mengedit kode sumber	Butuh mempelajari UI debugger IDE tertentu; berlebihan untuk pemeriksaan satu-nilai
Pernyataan assert	Mendokumentasikan dan memeriksa invarian internal yang TIDAK BOLEH pernah false bila kode benar	Nonaktif secara default saat runtime -- harus dijalankan dengan flag JVM -ea, atau assertion diam-diam dilewati

Biseksi: cara tercepat melokalisasi bug yang tidak diketahui

Ketika output program salah di suatu tempat dalam urutan langkah yang panjang tapi Anda belum tahu di mana, cetak (atau pasang breakpoint pada) nilai kira-kira di tengah urutan, bukan di awal sekali. Bila sudah salah di situ, bug ada di paruh pertama; bila masih benar, bug ada di paruh kedua. Mengulangi biseksi ini mempersempit pencarian seratus baris menjadi segelintir baris dalam sekitar tujuh pemeriksaan, ide pembagian-dua yang sama yang diformalkan Struktur Data (mata kuliah berikutnya) sebagai binary search.

Alat	Cocok untuk	Catatan
		assert untuk bug ANDA sendiri, tidak pernah untuk memvalidasi input pengguna assert firstName != null; tepat untuk mendokumentasikan "ini tidak boleh pernah null bila kode saya sendiri benar" -- dan karena assertion nonaktif secara default di produksi, kode yang bergantung pada assert benar-benar berjalan untuk menolak DATA buruk (berbeda dari menangkap bug di logika ANDA sendiri) akan diam-diam meloloskan data buruk itu begitu saja di deployment sungguhan. Input pengguna dan data tak terpercaya lainnya harus berada di balik pemeriksaan if eksplisit (atau, mulai Bab 6, exception yang dilempar), tidak pernah hanya di balik assert.

3.7 Contoh Kerja: Perhitungan Denda dan Status Perpanjangan untuk Book

Contoh kerja bab ini menambahkan dua method kecil ke Book: calculateFine(int daysOverdue), mendemonstrasikan aritmetika primitif int/double dan struktur tingkat if/else if/else, dan renewalStatusLabel(), menulis ulang keputusan tiga-arah yang persis sama yang sudah dikenal dari timesRenewed sebagai ekspresi switch.

Fase 1 — Understand

```
Kebutuhan baru: diberikan berapa hari sebuah buku terlambat, hitung
jumlah denda yang meningkat semakin lama buku itu tertahan, lalu
dibatasi maksimum -- plus label yang bisa dibaca manusia untuk berapa
kali buku diperpanjang, ditulis ulang memakai sintaks switch-ekspresi
baru bab ini.
```

Fase 2 — Abstract

```
Satu PERILAKU baru yang menerima int primitif dan mengembalikan
double primitif (jadwal denda yang monoton meningkat, lalu
dibatasi), plus PERILAKU kedua yang menyatakan ulang state yang
sudah ada (timesRenewed) lewat konstruksi alur-kontrol berbeda
namun setara.
```

Fase 3 — Design

```
FASE 3 -- DESIGN

Method baru (tanpa field baru -- keduanya membaca state yang
sudah ada atau parameter):
  calculateFine(int daysOverdue) -> double
    daysOverdue <= 0            -> 0.00
    daysOverdue dalam 1..7      -> daysOverdue * 0.50
    daysOverdue dalam 8..30     -> 3.50 + (daysOverdue - 7) * 1.00
    daysOverdue > 30            -> 50.00 (dibatasi)
  renewalStatusLabel() -> String
    switch (timesRenewed) { 0 -> ...; 1 -> ...; default -> ...; }
```

Fase 4 — Analyze

```
Argumen kebenaran: tiga perbandingan calculateFine (<= 0, <= 7,
<= 30) saling eksklusif dan kolektif menyeluruh -- setiap int jatuh
ke tepat satu cabang, dan jumlah denda tidak pernah menurun seiring
daysOverdue membesar, sesuai kebutuhan. Ekspresi switch
renewalStatusLabel menyeluruh berdasarkan konstruksi (default
mencakup setiap int yang tidak eksplisit terdaftar), sehingga tidak
bisa pernah gagal mengembalikan nilai seperti rantai if yang tidak
lengkap bisa diam-diam gagal.
```

Fase 5 — Implement

```
// field, constructor, dan getter: tak berubah dari Bab 2

public double calculateFine(int daysOverdue) {
    if (daysOverdue <= 0) {
        return 0.0;
    } else if (daysOverdue <= 7) {
        return daysOverdue * 0.50;
    } else if (daysOverdue <= 30) {
        return 3.50 + (daysOverdue - 7) * 1.00;
    } else {
        return 50.00;
    }
}

public String renewalStatusLabel() {
    return switch (timesRenewed) {
        case 0 -> "No renewals yet";
        case 1 -> "Renewed once";
        default -> "Max renewals reached";
    };
}
```

Driver di bawah mendemonstrasikan kedua method baru, beriterasi lewat sebuah array sampel jumlah hari terlambat dengan enhanced for-loop (Bagian 3.4), dan ditutup dengan perbandingan == vs .equals() yang diperingatkan Bagian 3.5, sengaja membangun String non-pool dengan new String(...) agar perbedaannya terlihat. Ditampilkan di sini secara utuh, persis seperti tampil di Code/Java/Chapter03/LibraryDemo.java, dan output lengkapnya persis apa yang diverifikasi Lampiran 3.A:

```
Book cleanCode = new Book("Clean Code", "Robert C. Martin",
    "978-0132350884", 2008);
System.out.println(cleanCode);

cleanCode.borrow();
cleanCode.renew();
System.out.println("Renewal status: " + cleanCode.renewalStatusLabel());

int[] overdueDaysSamples = {0, 3, 15, 45};
for (int days : overdueDaysSamples) {
    double fine = cleanCode.calculateFine(days);
    System.out.printf("Overdue %2d day(s) -> fine: %.2f%n", days, fine);
}

String isbnFromCatalog = "978-0132350884";
String isbnFromScanner = new String("978-0132350884");
System.out.println("== comparison:        "
    + (isbnFromCatalog == isbnFromScanner));
System.out.println(".equals() comparison: "
    + isbnFromCatalog.equals(isbnFromScanner));
```

Telusuri sendiri sebelum melanjutkan

Setelah borrow() dan satu panggilan renew(), timesRenewed bernilai 1 -- apa yang dikembalikan renewalStatusLabel()? Telusuri tingkatan denda: 0 hari persis 0.00 (batas milik cabang pertama, daysOverdue <= 0); 3 hari jatuh di tingkat 1-7 (3 * 0.50); 15 hari jatuh di tingkat 8-30 (3.50 + 8 * 1.00); 45 hari melebihi 30 dan dibatasi 50.00. Dan isbnFromScanner, dibangun dengan new String(...), dijamin menjadi objek berbeda dari literal ber-pool isbnFromCatalog -- jadi apa yang dicetak ==, dan apa yang dicetak .equals()?

3.8 Meng-compile dan Menjalankan Program Anda

Tidak ada yang berubah dari prosesnya: javac Book.java LibraryDemo.java, lalu java LibraryDemo. Ekspresi switch Java 14+ tidak butuh flag compiler khusus pada JDK mana pun mulai versi 21 (fitur ini difinalisasi sebagai fitur bahasa permanen di Java 14, jauh sebelum baseline JDK 21/25 mata kuliah ini).

```
$ javac Book.java LibraryDemo.java
$ java LibraryDemo
```

3.9 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi mata kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep dipakai dalam sistem sungguhan yang berjalan, tidak pernah logika bisnis proprietary atau kode sumber lengkap Profitina, yang tetap rahasia.

Proses rilis Profitina sendiri dijaga gerbang oleh rangkaian tes otomatis lebih dari 978 tes (pytest) yang harus lulus penuh sebelum perubahan kode apa pun mencapai server produksi -- persis disiplin yang

dilayani perbedaan assert-vs-input Bagian 3.6 dan teknik biseksi, diskalakan dari program satu mahasiswa ke produk sungguhan yang dipakai bisnis nyata. Tes yang gagal dalam rangkaian itu didiagnosis dengan alat inti yang persis sama yang diperkenalkan bab ini: memeriksa nilai tertentu pada titik tertentu, mempersempit di mana hasil tak terduga pertama kali muncul, dan tidak pernah memercayai asumsi yang belum benar-benar diperiksa.

Backend Profitina juga melakukan percabangan dan kategorisasi bertingkat substansial secara internal -- misalnya, memutuskan bagaimana menyajikan hasil AI-visibility sebuah bisnis melibatkan perbandingan nilai terhitung terhadap sejumlah kecil ambang batas, secara konseptual bentuk yang sama dengan tingkatan calculateFine bab ini, hanya diterapkan pada masalah berbeda. Dan karena mesin AI-visibility Profitina mem-parsing teks yang dihasilkan mesin jawaban AI eksternal untuk mencari penyebutan merek, ia terus-menerus dan dengan benar bergantung pada perbandingan string berbasis-nilai milik Python sendiri (Bab 3 jalur Python membahas ini secara langsung) -- tidak pernah pada perbandingan identitas string, yang ditunjukkan Bagian 3.5 Java bab ini sebagai pertanyaan berbeda yang mudah tertukar.

3.10 Ringkasan Bab dan Poin Penting

- Setiap nilai adalah referensi objek atau salah satu dari delapan tipe primitif Java; int dan double mencakup mayoritas besar pemakaian sungguhan.
- if/else if/else dan switch keduanya menyandikan percabangan; pilih switch ketika satu variabel dibandingkan terhadap sekumpulan kecil konstanta tepat, dan selalu pilih switch bentuk-panah modern, yang tidak bisa jatuh tembus.
- for cocok untuk jumlah iterasi yang diketahui, while memeriksa kondisinya sebelum iterasi pertama (bisa berjalan nol kali), do-while memeriksa setelahnya (selalu berjalan minimal sekali).
- String bersifat immutable dan (untuk literal) ber-pool; == membandingkan identitas dan .equals() membandingkan konten -- selalu pakai .equals() untuk membandingkan konten String.
- Debugging System.out, breakpoint IDE, dan assert masing-masing cocok untuk situasi berbeda; biseksi melokalisasi bug tak diketahui dengan cepat; assert mendokumentasikan invarian kode ANDA sendiri dan tidak pernah pengganti validasi input tak terpercaya.
- Book mendapat calculateFine(int) dan renewalStatusLabel() bab ini, tanpa menambah field baru -- kedua method adalah fungsi murni dari state yang sudah dimiliki Book.

Setiap alat yang diperkenalkan bab ini -- tipe primitif, percabangan, loop, perbandingan String, teknik debugging -- sudah ada di setiap program yang pernah Anda tulis sejauh ini, dipakai tanpa disebut namanya. Menamainya adalah yang memungkinkan Anda menalarkannya secara sengaja, bukan secara kebetulan.

3.11 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai. Kerjakan sebelum Lecture 4 (Kuis 1, mencakup Lecture 1-3).

1. Tulis ulang rantai if/else if/else calculateFine sebagai satu ekspresi tunggal memakai operator ternary bersarang (kondisi ? a : b). Apakah hasilnya lebih atau kurang terbaca dibanding versi if/else if/else? Justifikasi jawaban Anda.
2. Jelaskan, secara presisi, mengapa 0.1 + 0.2 == 0.3 bernilai false di Java, dan sebutkan cara yang benar untuk membandingkan dua nilai double untuk kesamaan "cukup dekat".
3. Apa yang dicetak System.out.println("exit" == "exit")? Apa yang dicetak System.out.println("exit" == new String("exit"))? Jelaskan perbedaannya memakai Gambar 3.4.
4. Tulis method singkat berbasis-loop countOverdueTiers(int[] daysArray) yang mengembalikan berapa banyak buku dalam array jatuh ke tingkat denda 8-30 hari. Bentuk loop mana (for, while, do-while, atau enhanced for) paling cocok, dan mengapa?
5. Bagian 3.6 mengatakan assert tidak boleh pernah memvalidasi input tak terpercaya. Tulis ulang baris ini agar benar menolak argumen daysOverdue negatif ke calculateFine bahkan ketika assertion nonaktif: assert daysOverdue >= 0;

Lampiran 3.A — Log Verifikasi Eksekusi

File Book.java dan LibraryDemo.java lengkap (Code/Java/Chapter03/, disediakan bersama buku ini) di-compile dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kode benar, bukan sekadar "terlihat benar". Output direproduksi verbatim di bawah.

```
$ javac Book.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
"Clean Code" by Robert C. Martin [ISBN 978-0132350884, 2008] - on the shelf
Renewal status: Renewed once
Overdue  0 day(s) -> fine: 0.00
Overdue  3 day(s) -> fine: 1.50
Overdue 15 day(s) -> fine: 11.50
Overdue 45 day(s) -> fine: 50.00
== comparison:        false
.equals() comparison: true
```

Referensi

[1] Oracle Corporation. "Primitive Data Types." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (diakses Agustus 2026).

[2] Oracle Corporation. "JEP 361: Switch Expressions." https://openjdk.org/jeps/361 (diakses Agustus 2026) — bentuk ekspresi switch, fitur bahasa permanen sejak Java 14.

[3] Oracle Corporation. "Java SE Support Roadmap." https://www.oracle.com/java/technologies/java-se-support-roadmap.html (diakses Agustus 2026).

[4] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008 (sumber judul contoh berjalan bab ini).

BAB 4 · BAB LATIHAN

Soal Latihan: Kelas, Field, Enkapsulasi & Dasar Bahasa

Tidak ada materi baru di sini -- hanya tujuh soal yang dirancang untuk memastikan Kuliah 1 sampai 3 benar-benar melekat, sebelum Kuis 1.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:

(1) Menjelaskan, dengan kata-kata Anda sendiri, apa itu kelas, objek, field, dan konstruktor, serta mengapa enkapsulasi penting. (2) Menulis kelas dari spesifikasi bahasa biasa: memilih field yang tepat, menulis konstruktor yang menetapkan semuanya, dan mengekspos state hanya lewat method. (3) Menulis method yang melakukan aritmetika primitif dengan benar, termasuk nilai yang harus dinormalisasi ke rentang tetap. (4) Menulis method yang mengembalikan instance baru dari kelasnya sendiri, bukan memutasi yang sudah ada. (5) Menulis dua kelas yang berkolaborasi -- method satu kelas memanggil method pada instance kelas lain.

4.1 Rekap: Kuliah 1–3 Secara Singkat

Kuliah 1 memperkenalkan pergeseran dari pemikiran prosedural ke pemikiran berorientasi objek -- membungkus data dan perilaku yang mengoperasikannya menjadi satu unit, sebuah kelas, alih-alih mengoper data mentah antar fungsi yang berdiri sendiri. Kuliah 2 memberi anatomi presisi pada ide itu: field menyimpan state sebuah objek, konstruktor menginisialisasi state itu tepat sekali saat pembuatan, dan enkapsulasi (field private, method public) adalah yang menjaga kode luar agar tidak bisa masuk dan merusak state itu secara langsung. Kuliah 3 kemudian melengkapi dasar-dasar bahasa Java di bawah semua itu -- tipe primitif, aturan aritmetika dan normalisasi yang mengaturnya, alur kontrol, String, dan alat debugging dasar yang Anda pakai ketika output program tidak sesuai harapan (Gambar 4.1).

Gambar 4.1 — Bab ini berada di Kuliah 4 dalam peta jalan 16-kuliah OOP: sebuah checkpoint, bukan topik baru, tepat sebelum Kuis 1.

4.2 Cara Menggunakan Bab Latihan Ini

> **Apa bab ini -- dan bukan apa**
>
> Ini adalah Bab Latihan: bab praktik, bukan bab materi baru. Setiap soal di bawah berakar pada materi spesifik dari Kuliah 1–3 (lihat Gambar 4.2). Setiap soal disertai PETUNJUK -- dorongan ke arah cara berpikir yang tepat -- tapi sengaja BUKAN solusi lengkap atau kode sumber. Mengerjakan soal itu sendiri, sekalipun tidak sempurna, mengajarkan jauh lebih banyak daripada membaca jawaban yang sudah jadi.

Asal Setiap Soal Latihan

Setiap soal di Bagian 4.3 dapat ditelusuri kembali ke salah satu dari tiga kuliah sebelumnya.

#	Soal	Berasal dari
1	Kalkulator Pecahan	K3 -- aritmetika int primitif, method static
2	Sudut Jam Analog	K3 -- aritmetika double, modulo, normalisasi
3	Kelas Akun Komputer	K2 -- field, konstruktor, enkapsulasi
4	Kelas Bilangan Kompleks	K2 -- field, method yang mengembalikan objek baru
5	Kelas Konversi Berat	K2 -- konstruktor, getter, nilai turunan
6	Kelas Pecahan Lanjut	K2/K3 -- toString(), method yang mengembalikan tipe kelasnya sendiri
7	Kelas Player dan Enemy	K2 -- dua kelas berkolaborasi, mutasi lewat method

Gambar 4.2 — Setiap soal di Bagian 4.3 dapat ditelusuri kembali ke salah satu dari tiga kuliah sebelumnya.

> **Sebelum Anda mulai**
>
> Coba setiap soal sungguh-sungguh dalam proyek .java baru sebelum membaca petunjuknya. Jika buntu, baca hanya petunjuknya -- bukan solusi -- lalu coba lagi. Ini persis disiplin yang akan dihargai oleh Kuis 1 di Bagian 4.4: kuisnya open-book, tapi itu bukan pengganti penalaran Anda sendiri.

4.3 Soal Latihan

4.3.1 Aritmetika Primitif & Alur Kontrol

Soal 1 ***[Mudah]***. Misalkan e1/d1 mewakili satu pecahan dan e2/d2 pecahan kedua, dengan e1 dan e2 bilangan bulat serta d1 dan d2 bilangan bulat positif. Tulis kelas Q11Fraction yang menghitung es/ds (jumlah) dan ep/dp (hasil kali) dari kedua pecahan, menggunakan rumus standar: es/ds = (e1*d2 + e2*d1) / (d1*d2), dan ep/dp = (e1*e2) / (d1*d2). Kedua hasil tidak perlu disederhanakan. Uji kelas Anda pada pasangan 1/2 & 1/3, 1/3 & 3/4, 1/2 & 2/3, dan 1/4 & 2/3.

> **Petunjuk**
>
> Soal sudah memberi Anda kedua rumus secara langsung -- tahan godaan untuk merancang objek Fraction penuh di sini (Soal 6 melakukan itu dengan benar); intinya soal ini adalah menerjemahkan es/ds dan ep/dp ke aritmetika int persis seperti yang diberikan, lalu menguji keempat pasangan.

Soal 2 ***[Mudah]***. Tulis kelas Q12Time dengan method static angleBetween(int hours, int minutes) yang menghitung sudut berlawanan arah jarum jam, dalam derajat bulat yang dinormalisasi ke 0-359, dari

jarum jam ke jarum menit pada jam analog tradisional. Ingat: jarum menit bergerak 6 derajat per menit (360/60), dan jarum jam merayap maju 0,5 derajat per menit selain 30 derajat per jam (360/12). Uji pada 9:00, 3:00, 18:00, 1:00, 2:30, dan 4:41.

> **Petunjuk**
>
> Hitung sudut jarum jam termasuk rayapan 0,5 derajat per menitnya, kurangi dengan sudut jarum menit, lalu normalisasi ke 0-359 dengan modulo double. Uji satu contoh yang BUKAN angka bulat (4:41) sebelum Anda percaya normalisasi Anda -- bug pembulatan-vs-pemotongan bisa bersembunyi di balik kasus angka bulat dan baru muncul di sana.

4.3.2 Field, Konstruktor & Enkapsulasi

Soal 3 *[Mudah]*. Definisikan kelas Q13ComputerAccount, dibangun dari tiga String: realName, userName, dan password. Implementasikan printRealName(), printUserName(), dan printPassword() (tanpa argumen), plus changePassword(String newPassword) (tanpa nilai kembali).

> **Petunjuk**
>
> Ini adalah kelas konstruktor-plus-method-akses paling sederhana yang mungkin: tiga field private, satu konstruktor yang menetapkan ketiganya, satu method per field untuk mencetaknya, dan satu method yang memutasi satu field. Tidak ada di sini yang seharusnya membutuhkan konsep baru di luar Kuliah 2.

Soal 4 *[Sedang]*. Definisikan Q14ComplexNumber, menyimpan bagian real dan bagian imajiner, dengan konstruktor dan getter. Implementasikan add, subtract, dan multiply (masing-masing mengembalikan Q14ComplexNumber baru), serta toString() yang menghasilkan "a + bi". Gunakan: (a+bi)+(c+di) = (a+c)+(b+d)i; (a+bi)-(c+di) = (a-c)+(b-d)i; (a+bi)*(c+di) = (ac-bd)+(ad+bc)i.

> **Petunjuk**
>
> add, subtract, dan multiply masing-masing menerima satu argumen Q14ComplexNumber dan harus mengembalikan Q14ComplexNumber BARU, bukan memodifikasi salah satu operand -- ketiga rumus aritmetika sudah diberikan langsung di soal, jadi kerja desain di sini sepenuhnya soal tipe kembalian, bukan matematikanya.

Soal 5 *[Mudah]*. Tulis Q15Weight(double pounds), menyimpan berat dalam pound, dengan getPounds() dan getKilograms() (menggunakan 1 pound = 0,45359237 kilogram).

> **Petunjuk**
>
> Simpan hanya nilai pounds sebagai field. getKilograms() harus menghitung konversi sesuai permintaan dari satu field itu, bukan menyimpan field kilogram kedua yang redundan yang bisa jadi tidak sinkron jika pounds pernah diubah.

4.3.3 Menggabungkan Semuanya

Soal 6 *[Sedang]*. Definisikan Q16Fraction dengan konstruktor, getNumerator/getDenominator, toString() (mis. "1/2"), serta getSum/getProduct, masing-masing mengembalikan Q16Fraction baru. Untuk f1 = new Q16Fraction(1,2) dan f2 = new Q16Fraction(3,7): f1.toString() harus mengembalikan "1/2"; f2.getProduct(f1) harus mencetak 3/14; f2.getSum(f1) harus mencetak 13/14.

> **Petunjuk**
>
> Ini adalah dua rumus Soal 1 lagi, tapi kali ini getSum dan getProduct masing-masing harus mengembalikan objek Q16Fraction yang utuh, bukan sepasang int mentah -- sehingga hasilnya bisa dijumlahkan atau dikalikan lagi tanpa perlu membongkar apa pun terlebih dahulu.

Soal 7 ***[Sedang]***. Definisikan dua kelas, Player dan Enemy, masing-masing dengan name, health, power, dan defense, plus konstruktor, getter, dan setter. Implementasikan attack(opposing) (pihak lawan menerima damage sebesar power pihak ini) dan takeDamage(opponentAttack) (health pihak ini berkurang sebesar opponentAttack - this.defense; jika health turun di bawah nol, cetak "{name} died!").

> **Petunjuk**
>
> attack() sebaiknya hanya menyerahkan SATU angka ke lawan -- power pihak ini -- dan biarkan takeDamage() milik lawan sendiri yang menerapkan defense-nya sendiri dan memutuskan apakah health sudah turun di bawah nol. Jangan menjangkau field lawan secara langsung dari dalam attack(); kedua kelas seharusnya hanya berbicara lewat method publiknya masing-masing.

4.4 Format Kuis 1: Open-Book, Individu, Berwaktu

Kuis 1 adalah penilaian take-home open-book, individu, dan berwaktu (sekitar 90 menit) yang mencakup persis tujuh jenis soal yang direview di bab ini -- tidak lebih. Penilaian dilakukan per-soal, bukan semua-atau-tidak-sama-sekali: nilai parsial diberikan untuk kelas yang kompilasi dan menangani kasus umum dengan benar sekalipun satu edge case terlewat.

Open-book berarti referensi, bukan kolaborator

Anda boleh berkonsultasi dengan buku teks, catatan Anda sendiri, dan slide kuliah saat menjawab. Anda TIDAK BOLEH berkolaborasi dengan teman sekelas atau mengumpulkan kode yang bukan benar-benar milik Anda sendiri -- kuis open-book menguji apakah Anda bisa MENERAPKAN apa yang sudah Anda pelajari, tanpa pengawasan.

Kriteria	Yang dinilai	Bobot
Kebenaran output	Apakah kelas menghasilkan hasil yang persis sesuai harapan untuk setiap kasus uji yang diberikan?	50%
Desain kelas & enkapsulasi	Apakah field bersifat private, dan state hanya bisa dijangkau lewat konstruktor dan method?	25%
Kejelasan kode	Apakah kelas mudah dibaca, penamaannya masuk akal, dan bebas dari kode mati atau duplikat?	15%
Kompilasi bersih	Apakah kelas berkompilasi dengan javac tanpa error dan tanpa warning?	10%

4.5 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Setiap satu dari tujuh soal bab ini mengikuti bentuk yang sama dengan model data backend Profitina sendiri pada skala yang jauh lebih besar: field private, konstruktor yang menginisialisasi semuanya secara penuh, dan method yang menjadi satu-satunya cara sah untuk membaca atau mengubah state itu dari luar kelas. Sebelum sebuah perubahan naik ke produksi di Profitina, engineer menjalankannya melalui checklist self-review singkat -- mirip sekali dengan Lampiran 4.A di bawah -- sebelum meminta pasang mata kedua: apakah perubahan itu menangani input persis yang dirancang untuknya, dan sudahkah diperiksa terhadap edge case yang mudah terlewat sekilas mata? Kebiasaan memverifikasi penalaran Anda sendiri sebelum review eksternal itulah yang membuat format kuis take-home open-book berfungsi, dan itulah tepatnya mengapa kasus 4:41 Soal 2 dan pengecekan health-di-bawah-nol Soal 7 keduanya penting jauh melampaui satu bab latihan ini.

4.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru -- ini adalah checkpoint yang mencakup Kuliah 1 sampai 3.
- Tujuh soal mencakup seluruh rentang yang direview: aritmetika primitif dan normalisasi (Kuliah 3), serta field, konstruktor, enkapsulasi, dan method yang mengembalikan objek baru (Kuliah 2).
- Setiap soal menyertakan petunjuk, bukan solusi -- mengerjakan penalaran dan kodenya sendiri adalah intinya.
- Kuis 1 adalah penilaian take-home open-book, individu, berwaktu: referensi diperbolehkan, tapi kodenya harus milik Anda sendiri, dan kebenaran pada setiap kasus uji yang diberikan adalah mayoritas nilai.

Kelas yang berkompilasi tidak sama dengan kelas yang benar -- bedanya selalu ada pada kasus uji yang tidak terpikirkan oleh Anda untuk dicoba.

Lampiran 4.A — Checklist Self-Check (Tidak Dinilai)

Sebelum mengumpulkan jawaban apa pun -- pada soal bab ini atau pada Kuis 1 itu sendiri -- jalankan lewat checklist ini. Butuh waktu beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah setiap kelas berkompilasi dengan javac tanpa error dan tanpa warning?
- Apakah setiap field bersifat private, dengan setiap nilai yang dibutuhkan pemanggil diekspos hanya lewat method?
- Apakah konstruktor setiap kelas menetapkan setiap field yang dideklarasikannya -- tidak ada field yang tertinggal di nilai default-nya secara tidak sengaja?
- Apakah method aritmetika Soal 4 dan 6 mengembalikan objek BARU alih-alih memutasi salah satu operand?
- Sudahkah Anda menguji setiap nilai contoh yang diberikan soal, bukan hanya yang kebetulan berupa angka bulat (kasus 4:41 Soal 2 sengaja ada untuk menangkap bug pembulatan yang tidak bisa diungkap oleh 9:00 dan 3:00)?
- Sudahkah Anda membaca ulang kode Anda sendiri seolah-olah Anda penilai skeptis yang mencari satu input yang bisa merusaknya?

Referensi

[1] Bab latihan ini dimodelkan dari Kuis 1 nyata mata kuliah ini (EF234302 Pemrograman Berorientasi Objek, tugas take-home open-book individu) -- tujuh soal yang sama, bobot poin yang sama.

[2] Oracle Corporation. "Classes." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html (diakses Agustus 2026).

BAB 5

Array, List & Kisah StringBuilder/StringBuffer

Setiap objek Book, Player, atau Fraction sejauh ini berdiri sendiri. Program nyata menyimpan banyak objek sekaligus -- seluruh rak buku sebuah perpustakaan, bukan satu buku saja. Bab ini memperkenalkan dua wadah Java untuk kebutuhan itu, jebakan klasik membangun String panjang sepotong demi sepotong, dan solusinya (Gambar 5.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Mendeklarasikan, membuat, dan mengindeks array berukuran tetap, serta menjelaskan tepatnya mengapa panjangnya tidak bisa berubah. (2) Mendeklarasikan ArrayList<T>, dan menggunakan add, remove, get, dan size untuk mengelola koleksi yang tumbuh. (3) Menjelaskan mengapa penggabungan String berulang dengan + di dalam loop bersifat kuadratik, dan memperbaikinya dengan StringBuilder.append(). (4) Menyebutkan satu perbedaan antara StringBuilder dan StringBuffer, dan menjelaskan mengapa StringBuilder adalah default yang benar di luar kode multi-thread. (5) Memperluas Pustaka dengan kelas Library yang mengelola koleksi objek Book, mengompilasinya, menjalankannya, dan memverifikasinya sendiri.

5.1 Dari Satu Buku ke Satu Rak Buku

Bab 1 hingga 4 bekerja dengan objek individual: satu Book, satu Fraction, satu Player. Setiap perpustakaan nyata, tentu saja, menyimpan banyak buku sekaligus, dan program yang hanya bisa menyimpan satu objek Book belum menjadi sistem perpustakaan sama sekali -- itu baru demonstrasi satu Book. Bab ini menutup celah tersebut dengan memperkenalkan dua cara Java untuk menyimpan koleksi objek, dan sekaligus bertemu dengan masalah kedua yang tampak tidak berhubungan -- membangun sepotong teks panjang sedikit demi sedikit -- yang ternyata berbagi jebakan biaya dasar yang sama persis, dan bentuk solusi yang sama persis.

Gambar 5.1 — Peta jalan 16 kuliah OOP. Bab ini adalah Kuliah 5, pemberhentian pertama setelah Kuis 1 dan awal bekerja dengan koleksi objek.

5.2 Array: Wadah Asli Java yang Berukuran Tetap

Array Java dideklarasikan dengan tipe dan tanda kurung siku, lalu dibuat dengan new dan panjang tetap yang tidak bisa berubah setelahnya: Book[] shelf = new Book[3]; membuat ruang untuk tepat tiga referensi Book, diindeks 0 hingga 2. Membaca atau menulis shelf[3] pada array tersebut -- satu

melewati panjangnya -- melempar ArrayIndexOutOfBoundsException saat runtime; Java tidak pernah diam-diam mengizinkannya dan tidak pernah mengubah ukuran array untuk mengakomodasinya.

Array adalah wadah level paling rendah yang ditawarkan Java: cepat, sederhana, dan didukung langsung oleh bahasa itu sendiri (sintaks tanda kurung siku itu bukan kelas yang Anda panggil method-nya). Satu keterbatasan seriusnya justru panjangnya yang tetap -- berguna ketika Anda tahu jumlah pastinya sebelumnya (papan catur selalu 8x8), kurang cocok ketika tidak, yang menggambarkan hampir setiap koleksi objek bisnis nyata, termasuk rak buku Pustaka.

Array Berukuran Tetap vs. List yang Bisa Tumbuh

Array Java dan ArrayList Java, serta satu-satunya list bawaan Python, dibandingkan pada properti yang penting sehari-hari.

Properti	Array Java (Book[])	ArrayList<Book> Java	List Python
Ukuran	`Tetap sejak dibuat`	`Tumbuh/menyusut otomatis`	`Tumbuh/menyusut otomatis`
Deklarasi + buat	`Book[] b = new Book[10];`	`List<Book> b = new ArrayList<>();`	`b = []`
Tambah elemen	`b[i] = book;  (i harus ada)`	`b.add(book);`	`b.append(book)`
Hapus elemen	`tidak didukung langsung`	`b.remove(book);`	`b.remove(book)`
Ukuran saat ini	`b.length`	`b.size()`	`len(b)`
Tipe elemen	`Tipe apa pun, termasuk primitif`	`Hanya objek (bukan int; pakai Integer)`	`Tipe apa pun, boleh campur`

Array Java adalah kontainer level paling rendah -- cepat dan sederhana, tapi panjangnya terkunci selamanya.
ArrayList membungkus array yang bisa diubah ukurannya secara internal; list Python dinamis secara default, tanpa saudara berukuran-tetap yang terpisah.

Gambar 5.2 — Array dan ArrayList Java dibandingkan dengan satu-satunya list bawaan Python. Perhatikan size() dan add() milik ArrayList terbaca hampir identik dengan len() dan append() milik Python.

5.3 ArrayList<T>: Koleksi yang Tumbuh Bersama Anda

java.util.ArrayList<T> menyelesaikan tepat masalah yang diangkat Bagian 5.2: List<Book> shelf = new ArrayList<>(); dimulai kosong dan tumbuh otomatis setiap kali shelf.add(book) dipanggil -- tidak ada panjang yang perlu dideklarasikan di awal dan tidak ada batas atas yang akan tercapai. <Book> dalam List<Book> adalah parameter tipe generik (dibahas penuh di Bab 10), untuk sekarang cukup dibaca sebagai "list ini menyimpan objek Book, dan hanya objek Book" -- mencoba shelf.add("bukan buku") adalah error kompilasi, bukan kejutan saat runtime.

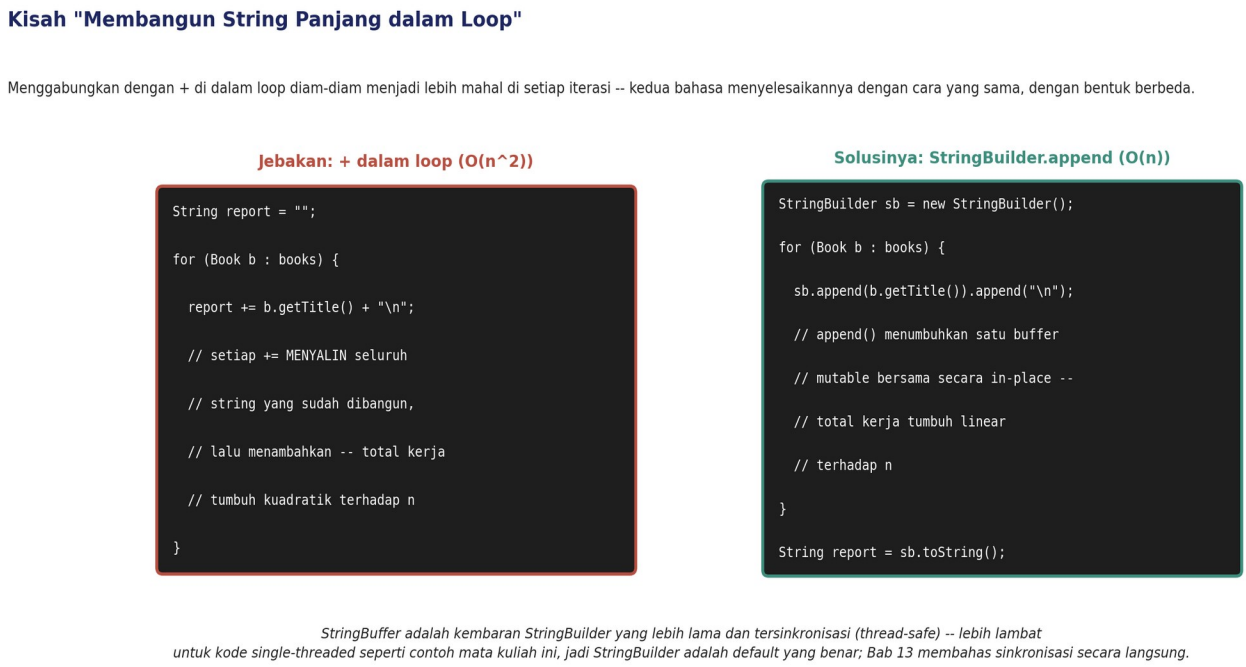
Operasi inti ArrayList, sekilas

add(item) menambahkan ke akhir; get(i) membaca item di indeks i; remove(item) atau remove(i) menghapus berdasarkan nilai atau indeks; size() melaporkan jumlah saat ini (tidak ada .length -- itu hanya ada pada array sejati). Loop for yang diperluas (for (Book b : shelf)) mengiterasi ArrayList persis seperti mengiterasi array biasa -- satu alasan lagi ArrayList hampir selalu menjadi pilihan default yang benar dibanding array mentah untuk koleksi objek.

> **ArrayList<int> tidak bisa dikompilasi**
>
> Generik Java hanya bekerja dengan tipe objek, tidak pernah dengan primitif -- List<int> adalah error kompilasi. Gunakan tipe wrapper yang di-boxing sebagai gantinya: List<Integer>, dan Java akan otomatis mengonversi ("autoboxing") antara int dan Integer sesuai kebutuhan. Inilah tepatnya situasi yang diperingatkan oleh baris "hanya objek" di Gambar 5.2.

5.4 Kisah StringBuilder/StringBuffer

String, seperti yang ditetapkan Bab 3, bersifat immutable: setiap += pada String tidak mengubahnya secara in-place tetapi membuat objek String yang sama sekali baru dan menyalin konten lama ke dalamnya. Di dalam loop yang berjalan n kali, itu berarti iterasi pertama menyalin string kecil, iterasi kedua menyalin string yang sedikit lebih panjang, dan seterusnya -- total kerja penyalinan tumbuh proporsional terhadap n^2 (kuadratik), bukan n, meskipun loop itu sendiri hanya berjalan n kali (Gambar 5.3).



Gambar 5.3 — Loop pembangunan laporan yang sama ditulis dengan cara yang mahal (String += dalam loop) dan cara yang benar (StringBuilder.append() dalam loop, dikonversi ke String hanya sekali di akhir).

StringBuilder menyelesaikan ini dengan benar-benar bersifat mutable: append() menumbuhkan satu buffer internal bersama secara in-place, sehingga total kerja di seluruh loop proporsional terhadap n, bukan n^2. Hanya setelah setiap potongan ditambahkan, .toString() dipanggil sekali, menghasilkan String akhir. Ini persis pola yang digunakan Library.generateReport() di Bagian 5.5 di bawah.

> **StringBuilder vs. StringBuffer**
>
> StringBuffer secara fungsional identik dengan StringBuilder -- method yang sama, perilaku yang sama -- kecuali setiap method-nya synchronized, artinya hanya satu thread yang boleh memanggilnya pada satu waktu. Sinkronisasi itu memakan biaya performa nyata dan tidak memberi keuntungan apa pun pada kode single-threaded biasa, yang menggambarkan setiap contoh dalam mata kuliah ini sejauh ini. StringBuilder karena itu adalah default yang benar; gunakan StringBuffer hanya ketika beberapa thread benar-benar berbagi satu builder, skenario yang dibahas langsung oleh Bab 13 (Konkurensi).

5.5 Memperluas Pustaka: Kelas Library

Library, diperkenalkan di bab ini, mengelola List<Book> dan mempraktikkan Bagian 5.3 dan 5.4 bersama-sama: addBook dan removeBook mengelola koleksi, dan generateReport membangun ringkasan multi-baris dari seluruh rak menggunakan StringBuilder alih-alih +=.

Fase 5 — Implement (Library.java, diringkas)

```
public class Library {
    private List<Book> books = new ArrayList<>();

    public void addBook(Book book) { books.add(book); }

    public boolean removeBook(String isbn) {
        for (int i = 0; i < books.size(); i++) {
            if (books.get(i).getIsbn().equals(isbn)) {
                books.remove(i);
                return true;
            }
        }
        return false;
    }

    public String generateReport() {
        StringBuilder sb = new StringBuilder();
        sb.append("Pustaka Library Report (");
        sb.append(books.size()).append(" book(s))\n");
        for (Book b : books) {
            sb.append("- ").append(b.getTitle());
            sb.append(" by ").append(b.getAuthor());
            sb.append(b.isOnLoan() ? " [on loan]" : " [on shelf]").append("\n");
        }
        return sb.toString();
    }
}
```

Perhatikan removeBook menggunakan loop for berindeks biasa, bukan ArrayList.removeIf berbasis lambda -- bentuk yang lebih singkat itu ada di Java modern, tetapi bergantung pada functional interface yang belum dibahas mata kuliah ini (Bab 7 dan seterusnya); loop biasa di atas get(i) sepenuhnya benar di sini dan hanya menggunakan alat yang sudah diperkenalkan.

5.6 Mengompilasi dan Menjalankan Program Anda

Tidak ada yang berubah dari prosesnya: javac Book.java Library.java LibraryDemo.java, lalu java LibraryDemo. java.util.ArrayList dan java.util.List masing-masing membutuhkan satu baris import di bagian atas file mana pun yang menggunakannya langsung (Library.java memiliki keduanya; LibraryDemo.java hanya menggunakan Library, jadi ia sendiri tidak membutuhkan import apa pun).

```
$ javac Book.java Library.java LibraryDemo.java
$ java LibraryDemo
```

5.7 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti pada setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, bukan logika bisnis atau kode sumber lengkap Profitina yang bersifat rahasia.

Backend Profitina secara rutin menyimpan koleksi yang ukurannya tidak pernah diketahui sebelumnya -- sekumpulan kata kunci yang dilacak sebuah bisnis, sekelompok respons mesin jawaban AI yang diambil untuk satu analisis -- persis situasi yang menurut Bagian 5.2 tidak bisa dilayani dengan baik oleh array berukuran tetap. Setiap koleksi itu didukung oleh struktur berukuran dinamis, list Python sendiri di backend (Bab 5 jalur Python membahas sisi ini secara langsung) yang secara konseptual mencerminkan ArrayList Java di sini. Dan karena lapisan pelaporan Profitina secara rutin menyusun teks yang lebih panjang -- ringkasan terformat dari hasil AI-visibility sebuah bisnis, misalnya -- ia mengikuti disiplin berbentuk-StringBuilder yang sama yang diperkenalkan Bagian 5.4: bangun potongan-potongannya dulu, gabungkan sekali, jangan pernah menggabungkan dalam loop.

5.8 Ringkasan Bab dan Poin-Poin Utama

- Array Java memiliki panjang tetap yang ditetapkan saat dibuat dan tidak pernah berubah; ArrayList<T> tumbuh dan menyusut otomatis dan hampir selalu menjadi default yang lebih baik untuk koleksi objek.
- Operasi inti ArrayList adalah add, remove, get, dan size -- tidak ada .length, yang merupakan sintaks khusus array.
- Generik (<Book> dalam List<Book>) hanya bekerja dengan tipe objek; gunakan wrapper yang di-boxing (Integer, bukan int) untuk list bilangan.
- String += di dalam loop bersifat kuadratik (O(n^2)) karena setiap += menyalin seluruh string yang sudah dibangun; StringBuilder.append() bersifat linear (O(n)) karena menumbuhkan satu buffer mutable bersama secara in-place.
- StringBuffer adalah kembaran StringBuilder yang tersinkronisasi -- benar hanya ketika beberapa thread berbagi satu builder; StringBuilder adalah default yang benar untuk single-threaded.
- Library mendapatkan addBook, removeBook, findByTitle, size, dan generateReport di bab ini -- Book itu sendiri tidak berubah sama sekali.

Setiap koleksi yang akan Anda kelola selama sisa mata kuliah ini -- rak sebuah perpustakaan, daftar pemain, sekelompok permintaan jaringan -- dibangun di atas persis dua gagasan yang diperkenalkan bab ini: wadah yang tumbuh bersama Anda, dan teks yang disusun sekali alih-alih disalin berulang-ulang.

5.9 Pertanyaan Tinjauan

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Mengapa Book[] shelf = new Book[5]; yang diikuti kemudian oleh shelf[5] = someBook; melempar exception, sementara shelf2.add(someBook) pada ArrayList<Book> berukuran 5 tidak pernah melakukannya?
2. Tulis ulang Library.generateReport() menggunakan String += alih-alih StringBuilder. Untuk perpustakaan dengan 10.000 buku, kira-kira berapa kali lebih banyak total kerja penyalinan karakter yang dilakukan versi ini dibandingkan versi StringBuilder? (Petunjuk: bandingkan n vs. n^2 untuk n = 10.000.)
3. Mengapa List<int> adalah error kompilasi di Java, dan apa cara yang benar untuk mendeklarasikan list bilangan bulat?
4. Tambahkan method Library.countOnLoan() yang mengembalikan berapa banyak buku dalam koleksi yang sedang dipinjam, menggunakan loop for yang diperluas dan getter isOnLoan() milik Book yang sudah ada.
5. Dengan kata-kata Anda sendiri, jelaskan mengapa StringBuffer akan menjadi pilihan yang benar alih-alih StringBuilder jika dua thread terpisah sama-sama menambahkan ke laporan yang sama pada saat yang sama. (Anda belum diharapkan menulis kode thread-safe -- Bab 13 membahas itu. Pertanyaan ini memeriksa apakah Anda memahami MENGAPA kedua kelas berbeda, bukan bagaimana menggunakan thread.)

Lampiran 5.A — Log Verifikasi Eksekusi

Berkas Book.java, Library.java, dan LibraryDemo.java yang lengkap (Code/Java/Chapter05/, disediakan bersama buku ini) telah dikompilasi dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kode benar-benar benar, bukan sekadar "tampak benar". Outputnya direproduksi persis di bawah ini.

```
$ javac Book.java Library.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
Fixed shelf (array), length = 2:
- Clean Code
- Effective Java

Library size after adding 3 books: 3
Pustaka Library Report (3 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on loan]
- The Pragmatic Programmer by Andrew Hunt [on shelf]
Removed The Pragmatic Programmer: true
Library size now: 2
Pustaka Library Report (2 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on loan]
```

Referensi

[1] Oracle Corporation. "Arrays." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html (diakses Agustus 2026).

[2] Oracle Corporation. "The ArrayList Class." The Java Tutorials. https://docs.oracle.com/javase/tutorial/collections/implementations/list.html (diakses Agustus 2026).

[3] Oracle Corporation. "Class StringBuilder" / "Class StringBuffer." Java SE 21 API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/StringBuilder.html (diakses Agustus 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017 (judul kedua yang berjalan berdampingan dengan "Clean Code" untuk contoh multi-buku bab ini).

BAB 6

Penanganan Exception & File I/O

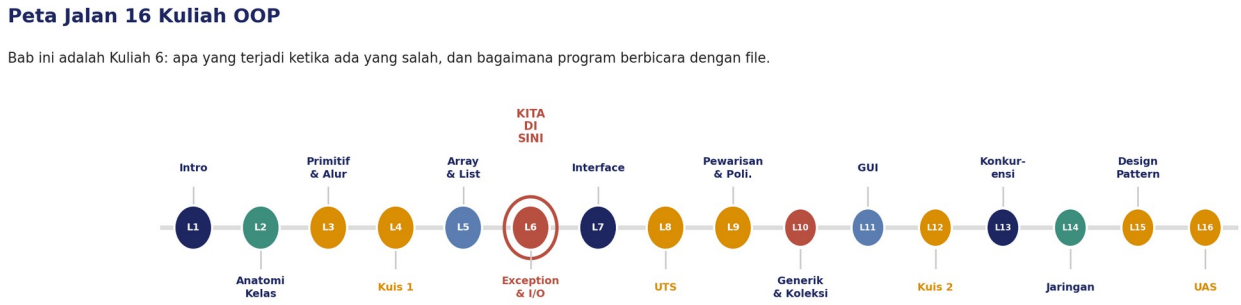
Setiap method sejauh ini mengasumsikan jalur mulus: input valid, file yang ada, ISBN yang benar-benar ada di perpustakaan. Program nyata tidak bisa berasumsi seperti itu. Bab ini memperkenalkan mekanisme exception Java untuk menangani kegagalan secara elegan, dan file I/O agar data Pustaka bisa bertahan lebih lama dari program yang membuatnya (Gambar 6.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan perbedaan antara checked dan unchecked exception, dan kapan Java memaksa Anda mendeklarasikan atau menangkapnya. (2) Menulis blok try-catch-finally, dan menyatakan dengan tepat kapan blok finally berjalan. (3) Mendefinisikan checked exception kustom dengan extends Exception, serta melempar dan menangkapnya dengan benar. (4) Membaca dan menulis file teks biasa menggunakan try-with-resources, dan menjelaskan mengapa try-with-resources lebih disukai daripada try-finally manual dengan close(). (5) Memperluas Pustaka dengan persistensi save/load dan method penghapusan yang lebih ketat yang melempar exception, mengkompilasinya, menjalankannya, dan memverifikasinya sendiri.

6.1 Dari Objek dalam Memori ke Program yang Bisa Gagal

Bab 1 sampai 5 membangun Pustaka yang bekerja sempurna -- selama tidak ada yang pernah salah. removeBook diam-diam mengembalikan false jika ISBN tidak ditemukan, dan setiap buku yang pernah dilihat program hanya hidup di memori: tutup programnya dan seluruh perpustakaan lenyap. Keduanya tidak bisa diterima dalam sistem nyata. Bab ini memperbaiki kedua celah itu: mekanisme exception Java memberi method cara yang prinsipil untuk menandakan "hal spesifik ini gagal," dan file I/O memberi Library cara untuk menyimpan buku-bukunya ke disk dan memuatnya kembali nanti.

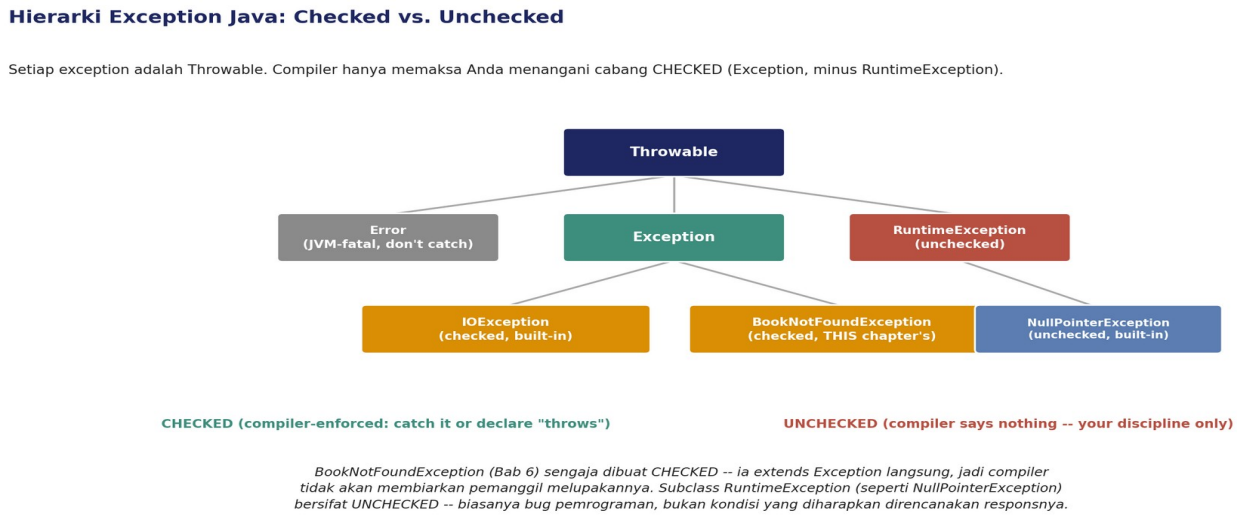


Gambar 6.1 — Peta jalan 16 kuliah OOP. Bab ini adalah Kuliah 6: apa yang terjadi ketika ada yang salah, dan bagaimana program berbicara dengan file.

6.2 Hierarki Exception Java: Checked vs. Unchecked

Setiap kondisi error di Java direpresentasikan oleh sebuah objek -- pada akhirnya, setiap kelas exception diturunkan dari Throwable. Error dan turunannya merepresentasikan kegagalan level-JVM (kehabisan memori, stack overflow) yang tidak diharapkan ditangkap oleh kode biasa. Exception dan turunannya merepresentasikan kondisi yang secara wajar diharapkan bisa ditangani program. Di dalam Exception, Java menarik satu garis penting lagi: RuntimeException dan turunannya

(NullPointerException, ArrayIndexOutOfBoundsException, dan sejenisnya) bersifat UNCHECKED -- compiler tidak pernah memaksa Anda menangkap atau mendeklarasikannya, karena biasanya menandakan bug pemrograman, bukan mode kegagalan yang diharapkan. Setiap subclass Exception lainnya bersifat CHECKED: compiler memaksa setiap pemanggil untuk menangkapnya atau mendeklarasikannya dengan throws, sehingga checked exception tidak pernah bisa diam-diam terlupakan (Gambar 6.2).



Gambar 6.2 — Hierarki exception Java. BookNotFoundException milik bab ini sengaja dibuat checked, sejajar dengan IOException bawaan yang juga checked.

> **Mengapa BookNotFoundException dibuat checked sama sekali?**
>
> Checked exception adalah alat yang tepat justru ketika kegagalan adalah hasil yang normal dan diharapkan, yang sungguh perlu direncanakan responsnya oleh pemanggil -- "ISBN yang Anda minta untuk dihapus tidak ada di perpustakaan ini" adalah persis kondisi semacam itu, bukan bug. Membuatnya checked berarti compiler sendiri tidak akan membiarkan maintainer kode ini di masa depan secara tidak sengaja memanggil removeBookOrThrow dan lupa bahwa method itu bisa gagal.

6.3 try, catch, dan finally

Blok try membungkus kode yang mungkin melempar; satu atau lebih blok catch masing-masing menangani satu tipe exception spesifik; blok finally opsional berjalan apa pun yang terjadi -- baik blok try selesai normal, melempar exception yang tertangkap, atau melempar yang tidak tertangkap dan akan menjalar lebih jauh ke atas call stack (Gambar 6.3).

Gambar 6.3 — Jalur alur kontrol try/catch/finally. Sifat khas finally adalah ia selalu dieksekusi.

try-catch-finally minimal

```
try {
    library.removeBookOrThrow(isbn);
} catch (BookNotFoundException e) {
    System.out.println("Menangkap exception yang diharapkan: " +
e.getMessage());
} finally {
    System.out.println("Ini selalu tercetak, exception atau tidak.");
}
```

Menangkap Exception (atau Throwable) hampir selalu salah

catch (Exception e) { } menelan setiap checked exception secara sembarangan -- termasuk yang tidak pernah Anda antisipasi dan tidak punya cara benar untuk menanganinya -- dan blok catch kosong diam-diam membuang kegagalan itu sepenuhnya, yang jauh lebih buruk daripada membiarkan program crash dengan stack trace yang jelas. Selalu tangkap tipe exception yang paling SPESIFIK yang kode Anda benar-benar bisa lakukan sesuatu yang berarti terhadapnya.

6.4 Exception Kustom: BookNotFoundException

Mendefinisikan exception kustom adalah pewarisan biasa: extends Exception (untuk checked exception, pilihan bab ini) atau RuntimeException (untuk yang unchecked), panggil super(message) di konstruktor, dan kelas baru itu adalah tipe exception yang sepenuhnya sah yang bisa dilempar dan ditangkap Java berdasarkan namanya.

Fase 6 — Implement (BookNotFoundException.java)

```
public class BookNotFoundException extends Exception {
    private final String isbn;

    public BookNotFoundException(String isbn) {
        super("No book found with ISBN " + isbn);
        this.isbn = isbn;
    }

    public String getIsbn() { return isbn; }
}
```

Library.removeBookOrThrow (Bagian 6.6) melempar exception ini ketika pencarian ISBN removeBook gagal, memberi pemanggil yang perlu tahu tentang kegagalan itu cara untuk memastikan mereka tidak bisa tidak sengaja mengabaikannya -- compiler yang menegakkannya.

6.5 File I/O dengan try-with-resources

Membaca atau menulis file membuka resource sistem (file handle) yang HARUS ditutup ketika Anda selesai menggunakannya, atau resource itu bocor -- masalah nyata dalam program yang berjalan lama yang membuka banyak file sepanjang masa hidupnya. Java 7 memperkenalkan try-with-resources khusus untuk membuat ini aman secara default: objek apa pun yang dideklarasikan di dalam tanda kurung try(...) yang mengimplementasikan AutoCloseable akan otomatis memanggil method close()-nya ketika blok try keluar, baik secara normal maupun lewat exception -- tidak perlu blok finally eksplisit.

try-with-resources vs. pola manual lama

```
// Modern (Java 7+): try-with-resources -- selalu utamakan ini
try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
    String line = reader.readLine();
    // reader.close() dipanggil otomatis di sini, terjamin
}

// Pola manual lama -- verbose dan mudah salah
BufferedReader reader = new BufferedReader(new FileReader(path));
try {
    String line = reader.readLine();
} finally {
    reader.close();  // mudah terlupakan, atau ditempatkan salah
}
```

IOException bersifat checked, dengan sengaja

Setiap method yang menyentuh filesystem bisa gagal karena alasan yang sepenuhnya di luar kendali program Anda -- file yang hilang, disk penuh, error izin akses -- jadi java.io.IOException adalah checked exception, dan Java memaksa setiap method yang membaca atau menulis file untuk menangkap IOException atau mendeklarasikan throws IOException. saveToFile dan loadFromFile di Bagian 6.6 keduanya mendeklarasikannya, menyerahkan tanggung jawab memutuskan cara bereaksi ke pemanggilnya sendiri.

6.6 Memperluas Pustaka: Persistensi dan Penghapusan yang Lebih Ketat

Library mendapat dua kemampuan terkait di bab ini: removeBookOrThrow, saudara yang lebih ketat dari removeBook (Bab 5) yang melempar BookNotFoundException alih-alih diam-diam mengembalikan false; serta saveToFile/loadFromFile, persistensi file teks sederhana yang dibangun di atas try-with-resources.

Fase 6 — Implement (Library.java, dipersingkat)

```
public void removeBookOrThrow(String isbn) throws BookNotFoundException {
    if (!removeBook(isbn)) {
        throw new BookNotFoundException(isbn);
    }
}

public void saveToFile(String path) throws IOException {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(path))) {
        for (Book b : books) {
            writer.write(b.getTitle() + "|" + b.getAuthor() + "|"
                + b.getIsbn() + "|" + b.getPublicationYear());
            writer.newLine();
        }
    }
}

public void loadFromFile(String path) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(path))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split("\\|", -1);
            books.add(new Book(parts[0], parts[1], parts[2],
                Integer.parseInt(parts[3])));
        }
    }
}
```

Setiap baris yang ditulis saveToFile adalah record sederhana yang dipisahkan pipe -- title|author|isbn|year -- sengaja dibuat sederhana daripada format serialisasi sungguhan (pekerjaan networking Bab 14 dan seterusnya akan memperkenalkan JSON), karena inti bab ini adalah disiplin penanganan exception dan manajemen resource, bukan format filenya.

6.7 Mengkompilasi dan Menjalankan Program Anda

Tidak ada yang berubah dari prosesnya: javac BookNotFoundException.java Book.java Library.java LibraryDemo.java, lalu java LibraryDemo. java.io.BufferedReader, BufferedWriter, FileReader, FileWriter, dan IOException masing-masing butuh satu baris import di bagian atas Library.java; LibraryDemo.java hanya butuh java.io.IOException, karena ia memanggil method Library alih-alih kelas I/O secara langsung.

```
$ javac BookNotFoundException.java Book.java Library.java LibraryDemo.java
$ java LibraryDemo
```

6.8 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Backend Profitina memanggil mesin jawaban AI eksternal dan API pihak ketiga secara terus-menerus -- panggilan yang bisa timeout, mengembalikan data yang salah bentuk, atau gagal sepenuhnya karena alasan yang sepenuhnya di luar kendali Profitina sendiri, persis situasi yang dijelaskan Bagian 6.2 sebagai alasan checked exception ada. Setiap panggilan semacam itu dibungkus dalam try-catch-nya sendiri dengan tipe exception yang spesifik dan bernama -- tidak pernah catch (Exception e) telanjang, persis karena alasan yang diperingatkan kotak jebakan Bagian 6.3 -- sehingga kegagalan sungguhan selalu terlihat dan ditangani secara sengaja, tidak pernah ditelan diam-diam. Dan karena backend Profitina menyimpan data jauh lebih tahan lama daripada file teks biasa bab ini (database sungguhan, bukan file teks -- Bab 14 dan mata kuliah Databases/FBP membahas mekanisme itu secara langsung), disiplin dasarnya identik: jangan pernah membiarkan resource terbuka lebih lama dari yang diperlukan, dan jangan pernah membiarkan kegagalan I/O lewat tanpa disadari.

6.9 Ringkasan Bab dan Poin-Poin Utama

- Setiap exception Java diturunkan dari Throwable; RuntimeException dan turunannya bersifat UNCHECKED (opsional bagi compiler), setiap subclass Exception lainnya bersifat CHECKED (ditegakkan compiler).
- try-catch-finally: finally selalu berjalan, baik blok try berhasil, melempar exception yang tertangkap, atau melempar yang menjalar tanpa tertangkap.
- Exception kustom adalah pewarisan biasa -- extends Exception untuk checked, RuntimeException untuk unchecked -- dan BookNotFoundException (milik bab ini) sengaja dibuat checked.
- Selalu tangkap tipe exception yang paling spesifik yang mungkin; jangan pernah membiarkan blok catch kosong, dan hindari menangkap Exception telanjang.
- try-with-resources otomatis menutup resource AutoCloseable apa pun (reader/writer file di sini) ketika blok try keluar, menggantikan pola manual try-finally-close() lama yang rawan salah.
- Library mendapat removeBookOrThrow, saveToFile, dan loadFromFile di bab ini -- Book sendiri sama sekali tidak berubah.

Program yang hanya pernah berjalan pada input sempurna, terhadap file yang selalu ada, belum menjadi program sungguhan -- itu adalah demonstrasi jalur mulus. Penanganan exception dan file I/O adalah yang memungkinkan Pustaka bertahan dari kontak dengan dunia nyata.

6.10 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Apakah NullPointerException checked atau unchecked? Jelaskan bagaimana Anda bisa mengetahuinya hanya dari hierarki kelasnya, tanpa mencarinya.
2. Tulis ulang contoh try-with-resources di Bagian 6.5 sebagai pola manual try-finally-close() lama, dan identifikasi satu cara konkret seorang maintainer bisa salah secara halus pada versi manual yang dibuat tidak mungkin oleh try-with-resources.
3. Mengapa Library.saveToFile mendeklarasikan throws IOException alih-alih menangkapnya secara internal dan mencetak pesan error?
4. Tambahkan method Library.countByAuthor(String author) yang mengembalikan berapa banyak buku dalam koleksi yang ditulis oleh penulis itu, dan putuskan (dengan justifikasi satu kalimat) apakah penulis yang tidak ditemukan seharusnya melempar exception atau cukup mengembalikan 0.
5. Dengan kata-kata Anda sendiri, jelaskan mengapa blok catch (Exception e) { } kosong dianggap salah satu kesalahan terburuk yang bisa dibuat program Java, meskipun ia berkompilasi dan berjalan tanpa error yang terlihat.

Lampiran 6.A — Log Verifikasi Eksekusi

File BookNotFoundException.java, Book.java, Library.java, dan LibraryDemo.java lengkap (Code/Java/Chapter06/, disediakan bersama buku ini) telah dikompilasi dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kodenya benar, bukan sekadar "terlihat benar". Output direproduksi verbatim di bawah.

```
$ javac BookNotFoundException.java Book.java Library.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
Library size after adding 3 books: 3
Caught expected exception: No book found with ISBN 0000000000000
Library size now: 2
Pustaka Library Report (2 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on shelf]
Saved library to library_export.txt
Reloaded library size: 2
Pustaka Library Report (2 book(s))
- Clean Code by Robert C. Martin [on shelf]
- Effective Java by Joshua Bloch [on shelf]
```

Referensi

[1] Oracle Corporation. "Exceptions." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/exceptions/ (diakses Agustus 2026).

[2] Oracle Corporation. "The try-with-resources Statement." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html (diakses Agustus 2026).

[3] Oracle Corporation. "Basic I/O." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/io/ (diakses Agustus 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Bab 10 ("Exceptions").

BAB 7

Interface & Kelas Abstrak

Sejauh ini, Pustaka hanya berisi satu jenis benda: Book. Perpustakaan nyata juga meminjamkan DVD, majalah, dan peralatan, dan tidak satu pun dari itu adalah buku. Bab ini memperkenalkan dua alat yang diberikan Java untuk berbagi struktur dan perilaku di antara kelas-kelas yang sungguh berbeda -- kelas abstrak, untuk tipe yang berbagi state dan kode sungguhan, dan interface, untuk kemampuan yang melintasi hierarki mana pun (Gambar 7.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan mengapa kelas abstrak tidak bisa diinstansiasi, dan menulis satu dengan method abstrak maupun method konkret. (2) Mendefinisikan interface Java, termasuk default method, dan menjelaskan bagaimana implements banyak interface berbeda dari extends satu kelas. (3) Menyatakan aturan praktis untuk memilih antara kelas abstrak dan interface untuk masalah desain tertentu. (4) Merefaktor kelas yang sudah ada agar extends superclass abstrak dan implements interface, tanpa merusak perilaku yang sudah ada. (5) Menulis loop yang memanggil method secara polimorfik di dua subtipe konkret yang tak terkait, mengkompilasinya, menjalankannya, dan memverifikasinya sendiri.

7.1 Rekap: Pustaka Sejauh Ini

Bab 1 sampai 6 membangun satu kelas Book yang terus bertambah kemampuannya dan Library yang mengelola koleksinya: field dan enkapsulasi (Bab 2), tipe primitif dan alur kontrol (Bab 3), array dan List (Bab 5), serta penanganan exception dengan persistensi file (Bab 6). Setiap bab itu memperbaiki Book atau Library tanpa pernah menanyakan pertanyaan yang lebih sulit: apa yang terjadi ketika Pustaka perlu menyimpan sesuatu yang sama sekali bukan Book?

Gambar 7.1 — Peta jalan 16 kuliah OOP. Bab ini adalah Kuliah 7: memberi kelas-kelas yang tak terkait sebuah kontrak bersama, tanpa berbagi implementasi.

7.2 Kelas Abstrak: State Bersama, Kode Bersama, Tanpa Instance

Kelas abstrak dideklarasikan dengan kata kunci abstract dan boleh berisi method abstrak (dideklarasikan tanpa implementasi -- setiap subclass konkret WAJIB meng-override-nya) maupun method konkret biasa (diimplementasikan sekali, diwarisi tanpa perubahan oleh setiap subclass). Java tidak akan membiarkan Anda menulis new pada kelas abstrak secara langsung -- hanya subclass

konkret yang telah mengimplementasikan setiap method abstrak yang bisa diinstansiasi. Ini adalah alat pilihan ketika subtipe sungguh berbagi field dan kode yang benar-benar berjalan, bukan sekadar konvensi penamaan.

Kelas abstrak minimal

```
public abstract class LibraryItem {
    protected final String title;
    protected boolean onLoan;

    // Method konkret -- setiap subclass mewarisi kode persis ini.
    public boolean isOnLoan() { return onLoan; }

    // Method abstrak -- setiap subclass WAJIB menyediakan versinya sendiri.
    public abstract double calculateFine(int daysOverdue);
}
```

Java hanya mengizinkan pewarisan tunggal

Sebuah kelas hanya boleh extends satu superclass, abstrak atau tidak -- subclass LibraryItem di bab ini (Book, DVD) tidak bisa juga extends kelas kedua. Ini persis celah yang diisi interface: sebuah kelas boleh implements sebanyak apa pun interface yang dibutuhkannya.

7.3 Interface: Sebuah Kemampuan, Bukan Anak Tangga

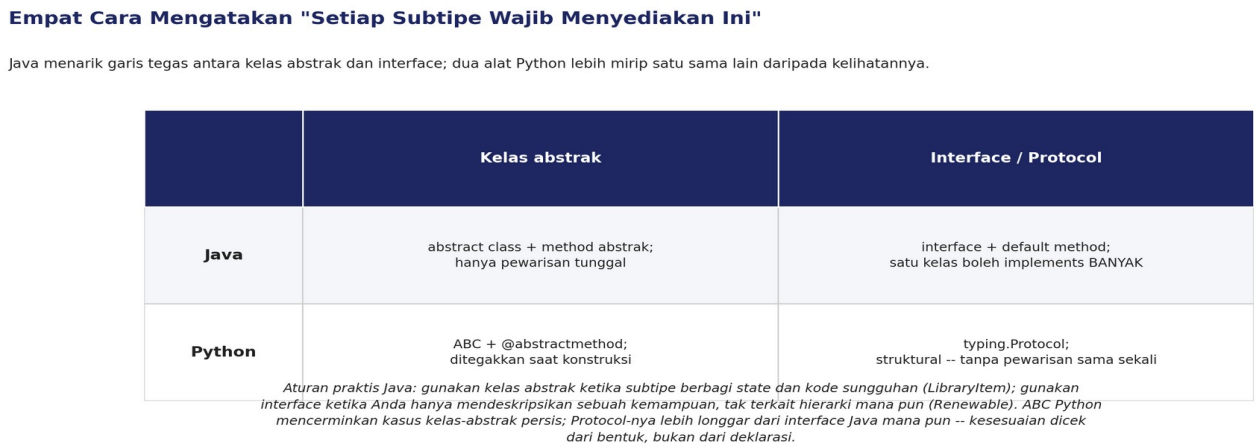
Interface mendeklarasikan sekumpulan method yang dijanjikan disediakan sebuah kelas, tanpa state (field) miliknya sendiri dan, secara tradisional, tanpa implementasi sama sekali -- hanya kontrak. Sebuah kelas ikut serta dengan implements, dan berbeda dari extends, sebuah kelas boleh implements sebanyak apa pun interface yang dibutuhkannya. Sejak Java 8, interface juga boleh berisi default method: sebuah method body lengkap yang diwarisi gratis oleh setiap kelas yang meng-implement-nya, kecuali ia memilih untuk meng-override-nya.

Interface minimal dengan default method

```
public interface Renewable {
    boolean renew();
    String renewalStatusLabel();

    // Default method (Java 8+): dibangun sepenuhnya dari dua method
    // di atas -- setiap kelas yang meng-implement mendapat ini gratis,
    // dan tetap boleh meng-override-nya jika perlu.
    default String renewalReceipt(boolean ok) {
        return "Renewal attempt: " + (ok ? "SUCCESS" : "DENIED")
            + " -- " + renewalStatusLabel();
    }
}
```

Perhatikan apa yang TIDAK disebutkan Renewable: tidak ada sebutan title, ISBN, atau apakah item sedang dipinjam. Ia hanya mendeskripsikan satu kemampuan sempit -- bisakah peminjaman benda ini diperpanjang? -- sepenuhnya independen dari jenis benda apa itu. Book dan DVD bisa sama-sama implements Renewable meski hampir tidak punya kesamaan lain, dan tipe item masa depan yang tidak pernah bisa diperpanjang (ensiklopedia referensi-saja, misalnya) cukup tidak implements sama sekali (Gambar 7.2).

7.4 Memilih Antara Kelas Abstrak dan Interface



Empat Cara Mengatakan "Setiap Subtipe Wajib Menyediakan Ini"

Java menarik garis tegas antara kelas abstrak dan interface; dua alat Python lebih mirip satu sama lain daripada kelihatannya.

	Kelas abstrak	Interface / Protocol
Java	abstract class + method abstrak; hanya pewarisan tunggal	interface + default method; satu kelas boleh implements BANYAK
Python	ABC + @abstractmethod; ditegakkan saat konstruksi	typing.Protocol; struktural -- tanpa pewarisan sama sekali

Aturan praktis Java: gunakan kelas abstrak ketika subtipe berbagi state dan kode sungguhan (LibraryItem); gunakan interface ketika Anda hanya mendeskripsikan sebuah kemampuan, tak terkait hierarki mana pun (Renewable). ABC Python mencerminkan kasus kelas-abstrak persis; Protocol-nya lebih longgar dari interface Java mana pun -- kesesuaian dicek dari bentuk, bukan dari deklarasi.

Gambar 7.2 — Empat cara mengekspresikan "setiap subtipe wajib menyediakan ini," lintas Java dan Python.

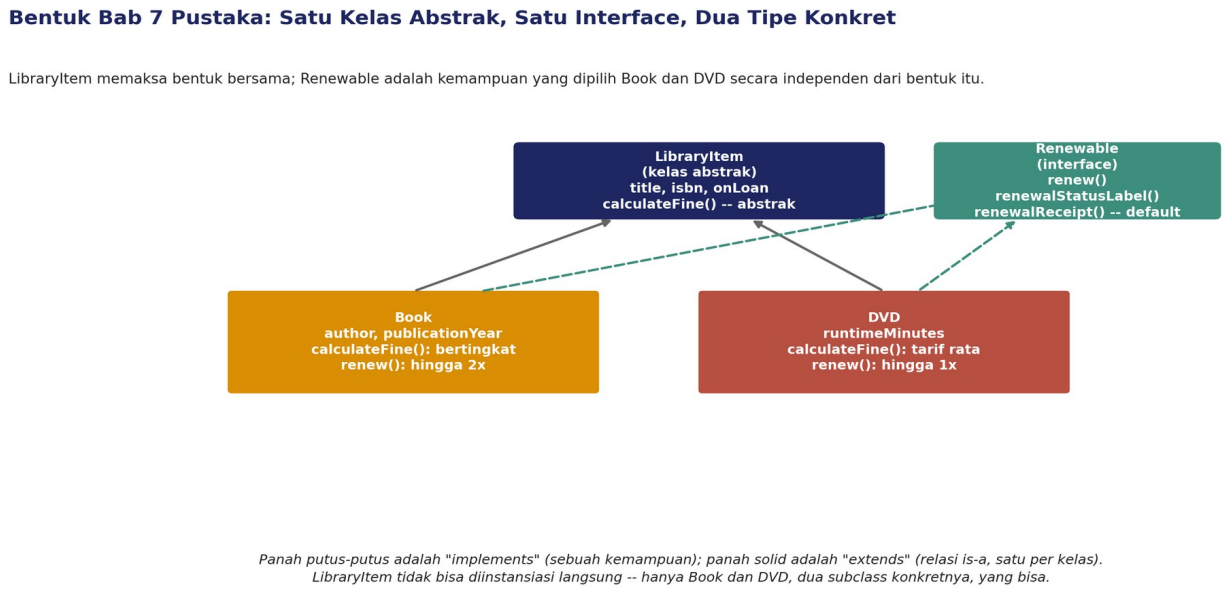
Aturan praktis: gunakan kelas abstrak ketika subtipe berbagi state dan perilaku sungguhan yang non-trivial yang jika tidak akan diduplikasi (field title, isbn, dan onLoan milik LibraryItem, ditambah logika borrow()-nya yang konkret). Gunakan interface ketika Anda mendeskripsikan kemampuan yang melintasi hierarki mana pun, dan beberapa kelas yang tak terkait mungkin ingin ikut serta (Renewable). Banyak kelas yang dirancang baik melakukan keduanya sekaligus -- persis yang dilakukan Book dan DVD di bab ini, extends LibraryItem SEKALIGUS implements Renewable.

7.5 Jawaban Python: ABC dan Protocol

Python tidak punya kata kunci interface atau kelas abstrak. Kelas dasar ABC milik modul abc ditambah decorator @abstractmethod adalah padanan terdekat kelas abstrak Java: kelas yang mewarisi ABC dan mendeklarasikan @abstractmethod tidak bisa diinstansiasi, dan Python melempar TypeError saat konstruksi jika subclass konkret lupa meng-override-nya -- salah satu dari sedikit tempat Python menegakkan kontrak sama sekali, alih-alih memercayai pemanggil seperti cara penanganan exception (Bab 6).

Untuk perilaku seperti interface, typing.Protocol milik Python (PEP 544) mengambil pendekatan STRUKTURAL yang sama sekali berbeda: sebuah kelas memenuhi Protocol hanya dengan memiliki method bernama dan bersignature sama -- tidak diperlukan deklarasi pewarisan jenis apa pun. Book dan DVD di jalur Python memenuhi Renewable murni dengan memiliki method renew() dan renewal_status_label(); tidak ada yang pernah menulis class Book(Renewable). Ini adalah filosofi duck-typing Python diterapkan secara formal: "jika ia berperilaku me-renew seperti Renewable, ia adalah Renewable (Gambar 7.3)."

7.6 Memperluas Pustaka: LibraryItem, Renewable, dan Tipe DVD Baru



Gambar 7.3 — Bentuk Bab 7 Pustaka: LibraryItem adalah superclass abstrak; Renewable adalah interface yang diikuti Book dan DVD masing-masing.

Field title, isbn, dan onLoan milik Book, beserta logika borrow()/returnItem()-nya, dipindahkan ke kelas abstrak baru, LibraryItem, yang juga mendeklarasikan calculateFine() sebagai abstrak -- setiap tipe item konkret wajib menyediakan aturan denda-terlambatnya sendiri, karena kebijakan keterlambatan buku dan kebijakan keterlambatan DVD sungguh berbeda aturannya, bukan sekadar berbeda angkanya. Book hanya menyimpan yang benar-benar khusus book: author, publicationYear, dan timesRenewed.

Fase 7 — Implement (LibraryItem.java, dipersingkat)

```
public abstract class LibraryItem {
    protected final String title;
    protected final String isbn;
    protected boolean onLoan;

    protected LibraryItem(String title, String isbn) {
        this.title = title;
        this.isbn = isbn;
        this.onLoan = false;
    }

    public boolean borrow() {
        if (onLoan) { return false; }
        onLoan = true;
        return true;
    }

    public abstract double calculateFine(int daysOverdue);

    public String describe() {
        return String.format("\"%s\" [ISBN %s] - %s",
            title, isbn, onLoan ? "ON LOAN" : "on the shelf");
    }
}
```

DVD adalah tipe item yang sama sekali baru di bab ini -- ia tidak berbagi apa pun dengan Book kecuali field LibraryItem dan kontrak Renewable. Ia bisa diperpanjang paling banyak sekali (Book mengizinkan dua kali perpanjangan) dan mengenakan tarif harian rata yang lebih tinggi tanpa batas atas (denda Book bertingkat setelah 30 hari).

Fase 7 — Implement (DVD.java, dipersingkat)

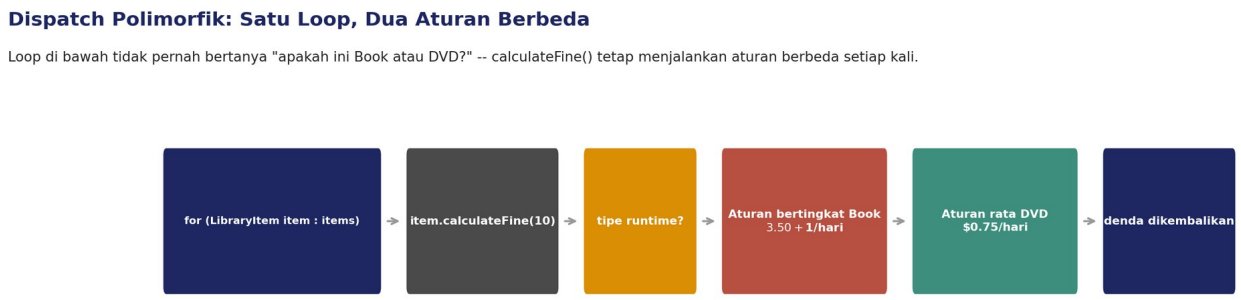
```
public class DVD extends LibraryItem implements Renewable {
    private final int runtimeMinutes;
    private int timesRenewed;
    private static final int MAX_RENEWALS = 1;
    private static final double DAILY_FINE = 0.75;

    public DVD(String title, String isbn, int runtimeMinutes) {
        super(title, isbn);
        this.runtimeMinutes = runtimeMinutes;
    }

    @Override
    public double calculateFine(int daysOverdue) {
        return daysOverdue <= 0 ? 0.0 : daysOverdue * DAILY_FINE;
    }

    @Override
    public boolean renew() {
        if (!onLoan || timesRenewed >= MAX_RENEWALS) { return false; }
        timesRenewed++;
        return true;
    }

    @Override
    public String renewalStatusLabel() {
        return timesRenewed >= MAX_RENEWALS ? "No renewals left" : "Renewable
once";
    }
}
```

borrow() milik Book meng-override method KONKRET yang diwarisi

Book meng-override borrow() milik LibraryItem sendiri -- bukan untuk mengulangi pemeriksaan "sudah dipinjam?", tetapi untuk menambahkan satu langkah khusus book (mereset timesRenewed) di atasnya, dengan memanggil super.borrow() terlebih dahulu. Meng-override method konkret untuk memperluasnya, bukan menggantinya sepenuhnya, sama sahnya dengan meng-override method abstrak.

7.7 Polimorfisme dalam Aksi: Program Driver

Hasil sungguhan bab ini adalah satu loop atas List<LibraryItem> yang berisi Book dan DVD sekaligus, di mana calculateFine(), renew(), dan describe() semuanya dipanggil secara polimorfik -- loop itu tidak pernah sekali pun bertanya "apakah ini Book atau DVD?" Java secara otomatis mengarahkan setiap panggilan ke override yang benar, berdasarkan tipe runtime objek yang sesungguhnya (Gambar 7.4).



Gambar 7.4 — Dispatch polimorfik: satu badan loop, dua aturan calculateFine() yang sungguh berbeda.

Fase 7 — Implement (LibraryDemo.java, dipersingkat)

```
List<LibraryItem> items = new ArrayList<>();
items.add(new Book("Clean Code", "Robert C. Martin", "9780132350884", 2008));
items.add(new DVD("The Imitation Game", "9900000000001", 114));

for (LibraryItem item : items) {
    item.borrow();
}

for (LibraryItem item : items) {
    System.out.printf("%s -> fine: $%.2f%n", item.describe(),
item.calculateFine(10));
}

for (LibraryItem item : items) {
    if (item instanceof Renewable r) {
        boolean ok = r.renew();
        System.out.println(item.getTitle() + ": " + r.renewalReceipt(ok));
    }
}
```

Library sendiri sengaja dibiarkan tidak berubah di bab ini -- ia masih mengelola List<Book> persis seperti sejak Bab 5. Menggeneralisasi Library agar bisa menyimpan LibraryItem mana pun adalah tugas Bab 9, setelah pewarisan dan polimorfisme mendapat bab penuh miliknya sendiri; program driver bab ini mendemonstrasikan polimorfisme secara langsung, dalam list mandiri, sehingga idenya tidak tercampur dengan logika persistensi file Library dari Bab 6.

7.8 Mengkompilasi dan Menjalankan Program Anda

javac LibraryItem.java Renewable.java Book.java DVD.java LibraryDemo.java, lalu java LibraryDemo. Tidak ada import baru yang dibutuhkan di luar yang sudah dipakai Bab 6.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java LibraryDemo.java
$ java LibraryDemo
```

7.9 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Backend Profitina berbicara dengan beberapa mesin jawaban AI eksternal yang berbeda, masing-masing dengan bentuk API, perilaku timeout, dan format respons sendiri. Alih-alih menulis kode pemanggilan terpisah yang terduplikasi untuk masing-masing, backend mendefinisikan sebuah interface bersama (persis pola bab ini -- kontrak bersama, tanpa implementasi bersama) yang diimplementasikan setiap konektor spesifik-provider, sehingga bagian lain sistem bisa memanggil salah satu dari mereka secara polimorfik tanpa peduli sedang berbicara dengan yang mana. Menambahkan provider AI baru nanti berarti menulis satu kelas baru yang meng-implement interface yang sudah ada, tanpa menyentuh kode mana pun yang sudah memanggilnya.

7.10 Ringkasan Bab dan Poin-Poin Utama

- Kelas abstrak tidak bisa diinstansiasi langsung; ia boleh mencampur method abstrak (setiap subclass wajib meng-override) dengan method konkret (diwarisi tanpa perubahan), dan sebuah kelas hanya boleh extends satu, abstrak atau tidak.
- Interface mendeklarasikan kemampuan tanpa state miliknya sendiri; sebuah kelas boleh implements sebanyak apa pun interface yang dibutuhkannya, dan sejak Java 8 interface boleh berisi default method dengan implementasi lengkap yang bisa diwarisi.
- Aturan praktis: kelas abstrak untuk subtipe yang berbagi state dan kode sungguhan; interface untuk kemampuan yang melintasi hierarki mana pun.
- ABC + @abstractmethod milik Python mencerminkan kelas abstrak Java dengan dekat; typing.Protocol bersifat struktural -- sebuah kelas memenuhinya hanya dari bentuknya, tanpa deklarasi pewarisan yang dibutuhkan.
- LibraryItem (abstrak) dan Renewable (interface) baru di bab ini; Book direfaktor untuk memakai keduanya, dan DVD adalah tipe konkret baru yang hampir tidak berbagi apa pun dengan Book kecuali dua kontrak ini.
- Satu loop atas List<LibraryItem> memanggil calculateFine(), renew(), dan describe() secara polimorfik di Book dan DVD -- badan loop tidak pernah memeriksa tipe konkret mana yang sedang dipegangnya.

Hierarki kelas yang hanya pernah menyimpan satu jenis objek belum pernah sungguh diuji -- begitu tipe kedua yang sungguh berbeda muncul, kelas abstrak dan interface adalah yang membuat kode lama tetap bekerja tanpa satu baris pun berubah.

7.11 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Mengapa Java tidak bisa menginstansiasi LibraryItem secara langsung dengan new LibraryItem("x", "y"), meskipun ia punya konstruktor?

2. Tambahkan tipe LibraryItem ketiga, Magazine, yang TIDAK Renewable sama sekali (majalah di perpustakaan ini hanya bisa dikembalikan, tidak pernah diperpanjang). Method LibraryItem mana yang tetap perlu diimplementasikan Magazine, dan interface mana yang benar untuk tidak diikutinya?
3. Tulis ulang default method renewalReceipt milik Renewable agar ia memanggil renew() sendiri alih-alih menerima hasilnya sebagai parameter. Jelaskan satu bug konkret yang muncul dari ini jika pemanggil pernah memanggil renewalReceipt() lebih dari sekali.
4. Dengan kata-kata Anda sendiri, jelaskan mengapa Book meng-implement Renewable sekaligus extends LibraryItem bukanlah sebuah kontradiksi, meskipun Java hanya mengizinkan pewarisan tunggal.
5. Andaikan List<Book> milik Library diubah hari ini menjadi List<LibraryItem> tanpa menunggu Bab 9. Sebutkan satu method Library yang sudah ada yang kodenya perlu berubah, dan jelaskan mengapa.

Lampiran 7.A — Log Verifikasi Eksekusi

File LibraryItem.java, Renewable.java, Book.java, DVD.java, dan LibraryDemo.java lengkap (Code/Java/Chapter07/, disediakan bersama buku ini) telah dikompilasi dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kodenya benar, bukan sekadar "terlihat benar". Output direproduksi verbatim di bawah.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
--- Overdue fines after 10 days (calculateFine) ---
"Clean Code" [ISBN 9780132350884] - ON LOAN -> fine: $6.50
"The Imitation Game" [ISBN 9900000000001] - ON LOAN -> fine: $7.50

--- Renewal attempts (Renewable) ---
Clean Code: Renewal attempt: SUCCESS -- Renewed once
The Imitation Game: Renewal attempt: SUCCESS -- No renewals left

--- Full item details (toString, type-specific) ---
"Clean Code" by Robert C. Martin [ISBN 9780132350884, 2008] - ON LOAN (renewed
1x)
"The Imitation Game" [ISBN 9900000000001, 114 min] - ON LOAN (renewed 1x)
```

Referensi

[1] Oracle Corporation. "Abstract Methods and Classes." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html (diakses Agustus 2026).

[2] Oracle Corporation. "Defining Methods in an Interface" dan "Default Methods." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html (diakses Agustus 2026).

[3] Python Software Foundation. "abc — Abstract Base Classes." Python 3 documentation. https://docs.python.org/3/library/abc.html (diakses Agustus 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Bab 20 ("Prefer interfaces to abstract classes").

BAB 8 · BAB LATIHAN

Soal Latihan: UTS — Meninjau Ulang Kuliah 1 Sampai 7

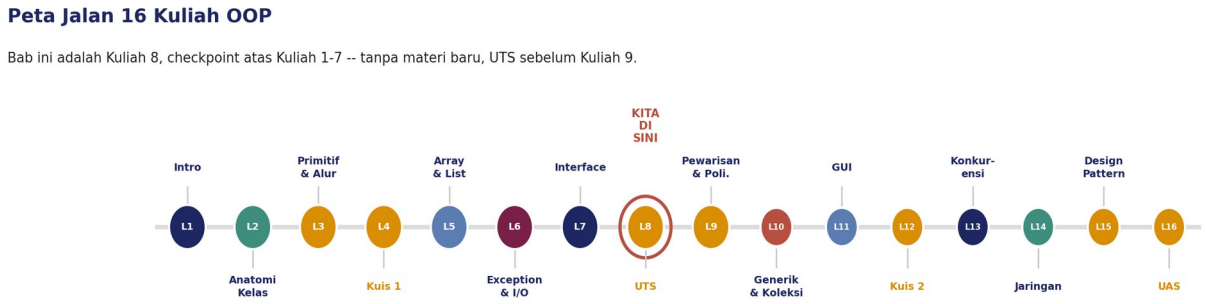
Tidak ada materi baru di sini -- delapan soal yang mencakup semuanya dari enkapsulasi hingga interface, dirancang untuk memastikan separuh pertama mata kuliah ini benar-benar melekat, sebelum UTS.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:

(1) Merancang kelas terenkapsulasi kecil yang menyimpan satu nilai dan menurunkan nilai lain darinya sesuai permintaan. (2) Menulis logika alur kontrol yang berperilaku benar pada setiap nilai batas, bukan hanya kasus yang jelas. (3) Membangun string terformat secara efisien dan memilih subset dari sebuah list sambil menjaga urutan aslinya. (4) Mendefinisikan dan melempar exception kustom, serta membaca dan menulis berkas teks polos dengan aman. (5) Merancang kelas abstrak dengan method konkret yang dibangun di atas method abstrak, dan interface dengan default method, lalu melakukan dispatch atas sebuah list bertipe interface itu secara polimorfik.

8.1 Rekap: Kuliah 1–7 Secara Singkat

Tujuh kuliah pertama membangun seluruh perkakas sehari-hari bahasa ini. Kuliah 1 memperkenalkan pergeseran dari pemikiran prosedural ke pemikiran berorientasi objek. Kuliah 2 memberi anatomi presisi pada ide itu -- field, konstruktor, dan enkapsulasi. Kuliah 3 melengkapi tipe primitif Java, alur kontrol, String, dan alat debugging dasar. Kuliah 4 adalah Kuis 1, checkpoint pertama atas semua itu. Kuliah 5 memperkenalkan array, koleksi List, dan kisah StringBuilder/StringBuffer untuk membangun teks secara efisien. Kuliah 6 menambahkan penanganan exception dan I/O berkas -- perkakas yang dibutuhkan program begitu ia harus bertahan dari input buruk dan hidup lebih lama dari satu kali eksekusi. Kuliah 7 memperkenalkan kelas abstrak dan interface, memungkinkan Book milik Pustaka dan tipe DVD yang sama sekali baru berbagi kontrak tanpa berbagi implementasi (Gambar 8.1).



Gambar 8.1 — Bab ini berada di Kuliah 8 dalam peta jalan 16-kuliah OOP: sebuah checkpoint, bukan topik baru, UTS sebelum Kuliah 9.

8.2 Cara Menggunakan Bab Latihan Ini

> **Apa bab ini -- dan bukan apa**
>
> Ini adalah Bab Latihan: bab praktik, bukan bab materi baru. Setiap soal di bawah berakar pada materi spesifik dari Kuliah 1–7 (lihat Gambar 8.2). Setiap soal disertai PETUNJUK -- dorongan ke arah cara berpikir yang tepat -- tapi sengaja BUKAN solusi lengkap atau kode sumber. Kedelapan soal ini berdiri sendiri, independen dari studi kasus Pustaka yang berjalan di seluruh bab biasa -- persis seperti soal-soal Kuis 1 di Bab 4.

Asal Setiap Soal Latihan

Setiap soal di Bagian 8.3 dapat ditelusuri kembali ke salah satu dari tujuh kuliah sebelumnya.

#	Soal	Berasal dari
1	Kelas Suhu	K2 -- field, konstruktor, nilai turunan
2	Konversi Nilai	K3 -- alur kendali, kondisi batas
3	Pemformat Daftar	K5 -- StringBuilder, array
4	Top-N Skor	K5 -- List, seleksi yang menjaga urutan
5	Cek Rentang Skor	K6 -- exception checked kustom
6	File Log Skor	K6 -- I/O berkas, try-with-resources
7	Gaji Karyawan	K7 -- kelas abstrak, method konkret di atas method abstrak
8	Item Berdiskon	K7 -- interface, default method, dispatch polimorfik

Gambar 8.2 — Setiap soal di Bagian 8.3 dapat ditelusuri kembali ke salah satu dari tujuh kuliah sebelumnya.

> **Sebelum Anda mulai**
>
> Coba setiap soal sungguh-sungguh dalam proyek .java baru sebelum membaca petunjuknya. Jika buntu, baca hanya petunjuknya -- bukan solusi -- lalu coba lagi. Ini persis disiplin yang akan dihargai oleh UTS di Bagian 8.4: ujiannya open-book, tapi itu bukan pengganti penalaran Anda sendiri.

8.3 Soal Latihan

8.3.1 Review Field, Konstruktor & Enkapsulasi

Soal 1 ***[Mudah]***. Definisikan kelas Q81Temperature(double celsius), hanya menyimpan nilai Celsius, dengan getCelsius(), getFahrenheit() (F = C * 9/5 + 32), dan getKelvin() (K = C + 273,15).

> **Petunjuk**
>
> Simpan hanya field celsius. Baik getFahrenheit() maupun getKelvin() harus menghitung hasilnya sesuai permintaan dari satu field itu, bukan menyimpan dua field tambahan yang bisa jadi tidak sinkron jika celsius pernah diubah -- persis disiplin yang sama dengan kelas Weight di Bab 4.

8.3.2 Review Primitif & Alur Kontrol

Soal 2 *[Mudah]*. Tulis kelas Q82GradeConverter dengan method static String letterGrade(double score) yang mengembalikan "A" untuk score >= 85, "B" untuk score >= 70, "C" untuk score >= 55, "D" untuk score >= 40, dan "E" jika tidak. Uji pada 85,0, 84,99, 70,0, 69,99, 55,0, 54,99, 40,0, dan 39,99.

> **Petunjuk**
>
> Urutkan perbandingan Anda dari batas tertinggi ke terendah dan kembalikan begitu satu cocok -- rantai if/else-if, bukan rantai if independen. Kasus 84,99/69,99/54,99/39,99 sengaja ada untuk menangkap kesalahan off-by-one antara >= dan > yang tidak akan pernah terungkap hanya dengan angka bulat.

8.3.3 Array, List & StringBuilder

Soal 3 *[Sedang]*. Tulis method static String formatRoster(String[] names) yang menggabungkan nama dengan ", " kecuali dua nama terakhir digabung dengan " and " -- mis. {"Ann", "Budi", "Citra"} menjadi "Ann, Budi, and Citra". Tangani dengan benar kasus 0, 1, 2, dan 3-atau-lebih nama, dan bangun hasilnya dengan StringBuilder, bukan konkatenasi String berulang.

> **Petunjuk**
>
> Tangani kasus 0/1/2-nama sebagai cabang short-circuit tersendiri lebih dulu -- keduanya tidak butuh loop sama sekali. Untuk 3 atau lebih, tambahkan setiap nama kecuali dua terakhir diikuti ", ", lalu tambahkan nama kedua-dari-terakhir, literal ", and ", dan akhirnya nama terakhir.

Soal 4 *[Sedang]*. Tulis method static List<Integer> topN(List<Integer> scores, int n) yang mengembalikan n nilai terbesar di scores, tetapi dalam urutan relatif ASLI dari list input, bukan diurutkan berdasarkan nilai. Contoh: scores = [70, 95, 60, 88, 99], n = 3 -> [95, 88, 99] (tiga nilai terbesar, 95/88/99, dipertahankan dalam urutan kemunculan aslinya).

> **Petunjuk**
>
> Jangan urutkan list yang Anda kembalikan -- urutkan SALINANNYA untuk menemukan nilai pada posisi cutoff (nilai terbesar ke-n), lalu telusuri list ASLI dalam urutan aslinya dan simpan setiap elemen yang melewati cutoff itu, berhati-hatilah dengan nilai duplikat agar Anda tidak menyimpan lebih dari n elemen ketika beberapa nilai sama dengan cutoff.

8.3.4 Penanganan Exception & I/O Berkas

Soal 5 *[Sedang]*. Definisikan kelas exception checked kustom InvalidScoreException, dan method static int validateScore(int score) throws InvalidScoreException yang mengembalikan score tanpa perubahan jika berada antara 0 dan 100 inklusif, dan selain itu melempar InvalidScoreException yang pesannya menyertakan nilai yang tidak valid.

> **Petunjuk**
>
> InvalidScoreException sebaiknya extends Exception (exception checked, persis seperti BookNotFoundException di Bab 6), dengan konstruktor yang menerima pesan dan meneruskannya ke super(message). validateScore() hanya butuh satu kondisi if yang mencakup kedua arah di luar rentang sekaligus.

Soal 6 ***[Sedang]***. Tulis dua method static: void appendScoreRecord(String path, String name, int score) throws IOException, yang menambahkan satu baris "name,score" ke berkas di path, dan List<String[]> readScoreRecords(String path) throws IOException, yang membaca kembali setiap baris dan mengembalikan masing-masing sebagai array String dua elemen yang dipisah pada koma.

> **Petunjuk**
>
> Gunakan try-with-resources dengan BufferedWriter yang dibuka dalam mode append untuk method pertama, dan BufferedReader untuk yang kedua -- persis pola yang dipakai saveToFile/loadFromFile di Bab 6 untuk Pustaka. Jangan lupa bahwa mode append tidak boleh menimpa baris-baris berkas yang sudah ada.

8.3.5 Interface & Kelas Abstrak

Soal 7 ***[Sulit]***. Definisikan kelas abstrak Q87Employee dengan field name, method abstrak double monthlyPay(), dan method KONKRET double annualPay() yang mengembalikan monthlyPay() * 12. Lalu definisikan dua subclass konkret: Q87SalariedEmployee (gaji bulanan tetap) dan Q87CommissionEmployee (jumlah bulanan dasar ditambah tarif komisi dikalikan angka penjualan bulanan).

> **Petunjuk**
>
> annualPay() sebaiknya berada di kelas abstrak itu sendiri, bukan di salah satu subclass -- ia sudah sepenuhnya bisa diimplementasikan dalam istilah monthlyPay() apa pun yang akhirnya disediakan tiap subclass, persis seperti LibraryItem.describe() di Bab 7 yang merupakan method konkret dibangun di atas calculateFine() yang abstrak.

Soal 8 ***[Sulit]***. Definisikan interface Discountable dengan method double discountedPrice() dan default method Java 8+ String describeDiscount() yang dibangun sepenuhnya dari discountedPrice(). Lalu definisikan Q88ClearanceItem(double originalPrice) (discountedPrice() = originalPrice * 0,5) dan Q88MemberItem(double originalPrice, double memberDiscountRate) (discountedPrice() = originalPrice * (1 - memberDiscountRate)), keduanya meng-implement Discountable. Terakhir, jumlahkan discountedPrice() atas sebuah List<Discountable> yang berisi campuran kedua tipe.

> **Petunjuk**
>
> describeDiscount() sebaiknya memanggil this.discountedPrice() sendiri dan memformat hasilnya -- ia tidak butuh field sendiri, persis seperti Renewable.renewalReceipt() di Bab 7. Loop penjumlahan sebaiknya ditulis sepenuhnya dalam istilah tipe interface Discountable, bentuk dispatch polimorfik yang sama dengan loop List<LibraryItem> di Bab 7 atas Book dan DVD.

8.4 Format UTS: Open-Book, Individu, Take-Home

UTS adalah penilaian take-home open-book dan individu yang mencakup persis delapan jenis soal yang direview di bab ini -- tidak lebih. Penilaian dilakukan per-soal, bukan semua-atau-tidak-sama-sekali: nilai parsial diberikan untuk kelas yang kompilasi dan menangani kasus umum dengan benar sekalipun satu edge case terlewat.

Open-book berarti referensi, bukan kolaborator

Anda boleh berkonsultasi dengan buku teks, catatan Anda sendiri, dan slide kuliah saat menjawab. Anda TIDAK BOLEH berkolaborasi dengan teman sekelas atau mengumpulkan kode yang bukan benar-benar milik Anda sendiri, dan setiap pengumpulan harus menyertakan Ikrar Integritas Akademik yang ditandatangani -- ujian take-home open-book menguji apakah Anda bisa MENERAPKAN apa yang sudah Anda pelajari, tanpa pengawasan.

Kriteria	Yang dinilai	Bobot
Kebenaran output	Apakah kelas atau method menghasilkan hasil yang persis sesuai harapan untuk setiap kasus uji yang diberikan, termasuk nilai batas?	45%
Desain kelas & interface	Apakah field bersifat private, state hanya bisa dijangkau lewat method, dan pemisahan abstrak/interface (Soal 7-8) digunakan dengan benar?	25%
Keamanan exception & I/O	Apakah exception Soal 5 membawa pesan yang berguna, dan apakah I/O berkas Soal 6 menggunakan try-with-resources dengan benar?	15%
Kejelasan kode & kompilasi bersih	Apakah kode mudah dibaca, penamaannya masuk akal, dan bebas dari kode mati -- serta berkompilasi dengan javac tanpa error?	15%

8.5 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Kedelapan soal bab ini, digabungkan, mendekati versi mini dari apa yang sungguh dibutuhkan satu fitur di Profitina: satu atau dua kelas terenkapsulasi kecil (Soal 1), logika yang diuji-batas dengan cermat (Soal 2), penanganan teks dan list yang efisien untuk sebuah laporan atau respons API (Soal 3-4), penanganan aman untuk input buruk dan state yang persisten (Soal 5-6), dan, ketika beberapa implementasi dari kemampuan yang sama perlu ditukar masuk-keluar -- seperti yang dilakukan adapter mesin-jawaban-AI milik Profitina -- sebuah interface atau kelas abstrak yang membiarkan bagian lain sistem tetap tidak peduli tipe konkret mana yang sedang dipegangnya (Soal 7-8). UTS yang hanya menguji satu dari kemampuan ini akan melewatkan betapa seringnya fitur nyata membutuhkan beberapa di antaranya bekerja bersama dalam satu potongan kode kecil yang sama.

8.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru -- ini adalah checkpoint yang mencakup Kuliah 1 sampai 7.
- Delapan soal berdiri sendiri mencakup seluruh rentang yang direview: enkapsulasi (K2), batas alur kontrol (K3), StringBuilder dan List (K5), exception kustom dan I/O berkas (K6), serta kelas abstrak dan interface (K7).
- Setiap soal menyertakan petunjuk, bukan solusi -- mengerjakan penalaran dan kodenya sendiri adalah intinya.
- UTS adalah penilaian take-home open-book, individu: referensi diperbolehkan, tapi kodenya harus milik Anda sendiri, kebenaran pada setiap kasus uji yang diberikan (termasuk nilai batas) adalah porsi terbesar nilai, dan Ikrar Integritas Akademik yang ditandatangani wajib disertakan pada setiap pengumpulan.

Kelas yang berkompilasi tidak sama dengan kelas yang benar -- dan kelas yang benar untuk angka bulat tidak sama dengan yang benar untuk nilai yang persis berada di batas.

Lampiran 8.A — Checklist Self-Check (Tidak Dinilai)

Sebelum mengumpulkan jawaban apa pun -- pada soal bab ini atau pada UTS itu sendiri -- jalankan lewat checklist ini. Butuh waktu beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah setiap kelas dan method berkompilasi dengan javac tanpa error dan tanpa warning?
- Apakah setiap field bersifat private, dengan setiap nilai yang dibutuhkan pemanggil diekspos hanya lewat method?
- Sudahkah Anda menguji nilai batas persis Soal 2 (84,99, 69,99, ...), bukan hanya angka bulat yang bisa menyembunyikan bug >= versus >?
- Apakah topN() pada Soal 4 mengembalikan nilai dalam urutan ASLI-nya, bukan diurutkan, dan tidak pernah lebih dari n elemen ketika beberapa nilai sama pada cutoff?
- Apakah pesan exception Soal 5 benar-benar menyertakan nilai yang tidak valid, dan apakah I/O berkas Soal 6 membuka stream-nya dengan try-with-resources alih-alih close() manual?
- Apakah method abstrak dan method konkret Soal 7 berada pada kelas abstrak yang SAMA, dengan method konkret memanggil yang abstrak alih-alih menduplikasi logika di setiap subclass?
- Sudahkah Anda membaca ulang kode Anda sendiri seolah-olah Anda penilai skeptis yang mencari satu input yang bisa merusaknya?

Referensi

[1] Format penilaian bab latihan ini (open-book, individu, take-home, dengan Ikrar Integritas Akademik yang ditandatangani) dimodelkan dari UTS nyata mata kuliah ini (EF234302 Pemrograman Berorientasi Objek). Delapan soal latihannya dirancang baru agar sesuai dengan cakupan Kuliah 1-7 yang sesungguhnya diajarkan pada pembaruan mata kuliah ini -- soal-soal ujian asli berpusat pada algoritma tree dan linked-list rekursif, materi yang dalam pembaruan ini sengaja ditempatkan pada mata kuliah Struktur Data, bukan OOP (lihat catatan cakupan Bab 1), sehingga menggunakan ulang soal itu di sini akan menguji materi yang tidak pernah diajarkan mata kuliah ini.

[2] Oracle Corporation. "Abstract Methods and Classes" dan "Defining Methods in an Interface." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/ (diakses Agustus 2026).

BAB 9

Pewarisan & Polimorfisme

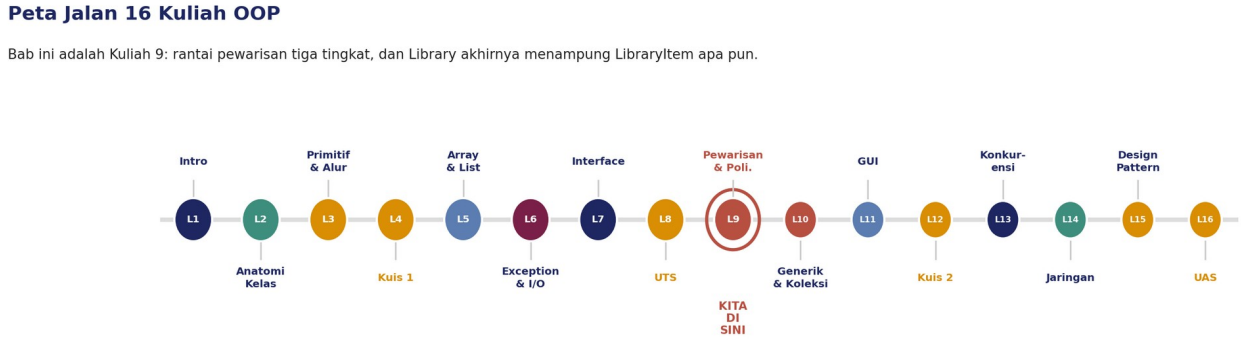
Bab 7 memperkenalkan LibraryItem, Renewable, dan tipe DVD yang baru, tetapi sengaja meninggalkan dua pertanyaan tanpa jawaban: seberapa dalam rantai pewarisan bisa sungguhan pergi, dan kapan Library sendiri berhenti bersifat khusus-Book? Bab ini menjawab keduanya -- menambahkan anak tangga ketiga ke hierarki, memformalkan apa arti polimorfisme sesungguhnya di level runtime, dan akhirnya menggeneralisasi Library agar menampung LibraryItem apa pun.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan perbedaan antara tipe statis (dideklarasikan) sebuah kelas dan tipe runtime-nya, serta mengapa pemanggilan method melakukan dispatch berdasarkan yang terakhir. (2) Menulis subclass yang extends kelas yang sudah konkret (bukan hanya kelas abstrak), dan memutuskan dengan benar apakah sebuah override sebaiknya memanggil super atau menggantikan perilaku warisan sepenuhnya. (3) Meng-override equals() dan hashCode() bersama-sama, dengan benar, dan menjelaskan mengapa meng-override hanya satu adalah bug meskipun compiler Java mengizinkannya. (4) Menjelaskan mengapa merekonstruksi koleksi polimorfik dari berkas datar membutuhkan tag tipe eksplisit, meskipun koleksinya sendiri bekerja secara polimorfik setelah dibangun. (5) Menggeneralisasi kelas koleksi yang khusus-tipe agar menampung supertipe bersama, memperbarui setiap method yang terpengaruh tanpa mengubah perilaku publiknya bagi pemanggil yang sudah ada.

9.1 Rekap: Pustaka Sejauh Ini

Bab 7 memberi Pustaka hierarki sungguhannya yang pertama: LibraryItem, kelas abstrak yang menyimpan title, isbn, dan onLoan plus calculateFine() abstrak yang wajib disediakan setiap subtipe konkret; Renewable, interface untuk kemampuan perpanjangan pinjaman; dan DVD, tipe konkret baru yang tidak berbagi apa pun dengan Book kecuali kedua kontrak itu. Bab 8 adalah checkpoint, bukan materi baru. Library, sementara itu, diam-diam tetap seperti yang ditinggalkan Bab 5 -- sebuah List<Book>, tidak mampu menampung DVD yang baru diperkenalkan Bab 7. Bab ini adalah saat itu akhirnya berubah.



Gambar 9.1 — Peta jalan 16-kuliah OOP. Bab ini adalah Kuliah 9: rantai pewarisan tiga tingkat, dan Library akhirnya menampung LibraryItem apa pun.

9.2 Pewarisan, Secara Formal: extends, super, dan Kontrak Override

Kelas yang extends kelas lain mendeklarasikan relasi is-a: setiap Book is-a LibraryItem, dan (sejak bab ini) setiap ReferenceBook is-a Book, yang membuatnya secara transitif juga LibraryItem. Pernyataan pertama konstruktor subclass adalah pemanggilan eksplisit ke super(...), meneruskan argumen ke atas rantai, atau pemanggilan implisit ke konstruktor tanpa-argumen milik superclass jika tidak ada yang dituliskan -- tidak ada opsi ketiga. Di dalam method yang meng-override, super.namaMethod(...) memanggil versi spesifik yang didefinisikan superclass, persis seperti Book.borrow() (Bab 7) menggunakan kembali pemeriksaan "sudah dipinjam?" milik LibraryItem sebelum menambahkan langkah khusus book-nya (Gambar 9.1).

Book.borrow() -- override yang memanggil super (Bab 7, tidak berubah)

```
@Override
public boolean borrow() {
    if (!super.borrow()) { return false; }
    timesRenewed = 0;
    return true;
}
```

Meng-override tidak mengharuskan memanggil super

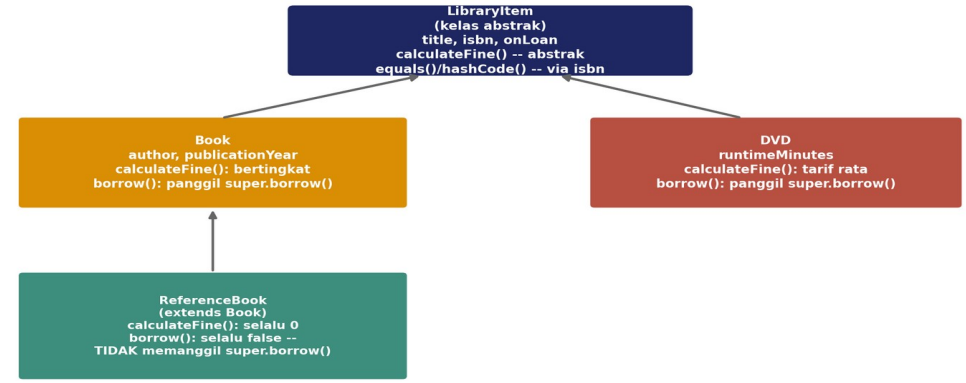
Tidak ada apa pun pada @Override atau mekanisme override itu sendiri yang memaksa sebuah method memanggil versi induknya -- memanggil super adalah pilihan yang dibuat override ketika ingin memperluas, bukan menggantikan, perilaku warisan. Bagian 9.3 menunjukkan pilihan sebaliknya pada hierarki yang sama persis.

9.3 Anak Tangga Ketiga: ReferenceBook dan Pewarisan Multi-Tingkat

ReferenceBook extends Book secara langsung, bukan LibraryItem -- menjadikan LibraryItem → Book → ReferenceBook rantai tiga tingkat yang sungguhan, pertama kalinya studi kasus mata kuliah ini melewati satu tingkat. ReferenceBook mewarisi setiap field dan method milik Book, termasuk yang tidak akan pernah dipakainya (timesRenewed, dan kata-kata bertingkat yang dihasilkan renewalStatusLabel()), dan meng-override tepat empat method untuk menegakkan satu aturan: item referensi tidak pernah meninggalkan gedung (Gambar 9.2).

Bentuk Bab 9 Pustaka: Tiga Tingkat Pewarisan

ReferenceBook extends Book, yang extends LibraryItem -- rantai yang dimulai Bab 7 kini punya tautan ketiga yang sungguhan.



Panah solid adalah "extends." ReferenceBook meng-override borrow()/renew() agar selalu menolak, TANPA memanggil versi super yang diwarisinya -- sebuah override tidak pernah wajib membangun di atas perilaku induknya; kadang override yang benar justru menggantikannya sepenuhnya.

Gambar 9.2 — Bentuk Bab 9 Pustaka: LibraryItem → Book → ReferenceBook, tautan ketiga yang sungguhan dalam rantai yang dimulai Bab 7.

ReferenceBook.java (dipersingkat -- keempat override ditampilkan)

```
public class ReferenceBook extends Book {
    public ReferenceBook(String title, String author, String isbn, int
publicationYear) {
        super(title, author, isbn, publicationYear);
    }

    // Sengaja TIDAK memanggil super.borrow() -- menggantikannya sepenuhnya.
    @Override
    public boolean borrow() { return false; }

    @Override
    public boolean renew() { return false; }

    @Override
    public String renewalStatusLabel() { return "Reference only -- cannot be
renewed"; }

    @Override
    public double calculateFine(int daysOverdue) { return 0.0; }
}
```

Field protected menjangkau setiap tingkat, bukan cuma satu

title dan isbn dideklarasikan protected pada LibraryItem, dua tingkat di atas ReferenceBook -- akses protected diwarisi secara transitif, sehingga ReferenceBook.toString() bisa membaca title dan isbn secara langsung, persis seperti Book, tanpa LibraryItem perlu memberikan akses itu kedua kalinya.

9.4 Polimorfisme, Secara Formal: Tipe Statis vs. Tipe Runtime

Variabel yang dideklarasikan sebagai LibraryItem item memiliki tipe statis LibraryItem, tetap selamanya pada saat kompilasi -- tetapi item bisa memegang Book, DVD, atau ReferenceBook pada saat runtime, dan itulah tipe dinamis (atau runtime) milik item. Ketika item.calculateFine(10) dipanggil, Java tidak melihat tipe statis untuk memutuskan calculateFine() mana yang dijalankan; Java melihat tipe runtime objek yang SESUNGGUHNYA dan melakukan dispatch ke sana, setiap saat. Inilah dynamic dispatch, dan inilah mekanisme di balik setiap loop polimorfik yang ditulis mata kuliah ini sejak Bab 7.

Menetapkan Book atau DVD ke variabel bertipe LibraryItem (upcasting) selalu aman dan terjadi otomatis -- sebuah Book memang benar-benar sebuah LibraryItem, tidak ada informasi yang hilang. Arah sebaliknya (downcasting, memperlakukan LibraryItem yang Anda duga sebenarnya DVD sebagai DVD) tidak otomatis dan tidak selalu aman: Java membutuhkan cast eksplisit, yang melempar ClassCastException saat runtime jika dugaan Anda salah, atau bentuk pattern-match instanceof yang digunakan di seluruh buku ini, yang memeriksa lebih dulu dan hanya melanjutkan jika dugaannya benar.

Upcasting otomatis; downcasting butuh pemeriksaan

```
LibraryItem item = new DVD("The Imitation Game", "9900000000001", 114); // upcast, otomatis

if (item instanceof DVD d) {          // downcast, diperiksa lebih dulu
    System.out.println(d.getRuntimeMinutes() + " minutes");
}
```

9.5 Setiap Kelas Diam-Diam adalah Subclass Object: equals() dan hashCode()

Setiap kelas di Java -- entah dinyatakan atau tidak -- pada akhirnya extends java.lang.Object, dan mewarisi equals() dan hashCode() default milik Object: perbandingan berbasis identitas, setara dengan == dan dengan alamat memori objek. Dua objek Book yang dibangun terpisah dari title, author, dan ISBN yang persis sama TIDAK .equals() di bawah default itu, yang jarang menjadi hal yang sungguh ingin diketahui program nyata. LibraryItem meng-override keduanya di bab ini agar membandingkan via isbn -- mengubah "apakah ini objek yang sama?" menjadi "apakah ini item dunia-nyata yang sama?"

Kesetaraan Identitas vs. Kesetaraan Nilai, Java dan Python Berdampingan

Setiap kelas di kedua bahasa mewarisi default berbasis identitas -- LibraryItem meng-override-nya di kedua jalur agar membandingkan via ISBN.

	Default warisan	Override LibraryItem
Java	Object.equals(): true hanya jika objek yang sama (seperti ==)	equals()/hashCode() via isbn; dibangun di atas Book(...).equals(Book(...))
Python	object.__eq__: true hanya jika objek yang sama (seperti is)	__eq__/__hash__ via isbn; mendefinisikan __eq__ saja membuat kelas TIDAK BISA di-hash -- __hash__ juga harus didefinisikan, atau diam-diam menjadi None

Kedua bahasa menegakkan aturan yang sama untuk alasan yang sama: hash sebuah objek HARUS dibangun dari field yang sama persis dengan yang dibandingkan equals()/__eq__, atau objek yang setara bisa jatuh ke bucket hash berbeda dan diam-diam hilang dari lookup set/dict.

Gambar 9.3 — Kesetaraan identitas vs. kesetaraan nilai, Java dan Python berdampingan.

LibraryItem.equals() dan hashCode() (dipersingkat)

```java
@Override
public boolean equals(Object other) {
    if (this == other) { return true; }
    if (!(other instanceof LibraryItem that)) { return false; }
    return this.isbn.equals(that.isbn);
}

@Override
public int hashCode() {
    return isbn.hashCode();
}
```

Java tidak akan menghentikan Anda meng-override hanya salah satunya

Kontraknya absolut -- jika a.equals(b) true, maka a.hashCode() harus sama dengan b.hashCode() -- tetapi compiler Java tidak menegakkan apa pun dari itu. Meng-override equals() saja tetap terkompilasi bersih dan berjalan, sampai suatu saat dua item yang "setara" jatuh ke bucket berbeda pada HashSet atau HashMap dan sebuah lookup yang seharusnya berhasil diam-diam tidak mengembalikan apa-apa. (Runtime Python lebih ketat di sini: mendefinisikan __eq__ saja membuat kelas tidak bisa di-hash secara default -- lihat Gambar 9.3.)

9.6 Menggeneralisasi Library: Satu Koleksi, LibraryItem Apa Pun

Field private milik Library berubah dari List<Book> menjadi List<LibraryItem> di bab ini, dan setiap method yang dibangun di atasnya -- addItem (dulu addBook), removeItem, removeItemOrThrow, findByTitle, generateReport -- kini bekerja lintas Book, DVD, dan ReferenceBook secara seragam, tanpa percabangan khusus-tipe pada satu pun di antaranya. Satu tempat kode khusus-tipe tetap sungguh diperlukan adalah persistensi berkas: satu baris teks polos tidak membawa informasi tipe apa pun, sehingga saveToFile/loadFromFile membutuhkan tag tipe eksplisit sebagai field pertama tiap baris.

Library.java -- addItem dan generateReport yang digeneralisasi (dipersingkat)

```
public class Library {
    private List<LibraryItem> items = new ArrayList<>();

    public void addItem(LibraryItem item) { items.add(item); }

    // Polimorfik dengan sengaja: tidak pernah memeriksa tipe konkret yang
dipegangnya.
    public String generateReport() {
        StringBuilder sb = new StringBuilder();
        for (LibraryItem item : items) {
            sb.append("- ").append(item.describe()).append("\n");
        }
        return sb.toString();
    }
}
```

Library.java -- format berkas dengan tag tipe (dipersingkat)

```
private String toLine(LibraryItem item) {
    if (item instanceof ReferenceBook rb) {
        return "REFBOOK|" + rb.getTitle() + "|" + rb.getAuthor() + "|" +
rb.getIsbn();
    } else if (item instanceof Book b) {
        return "BOOK|" + b.getTitle() + "|" + b.getAuthor() + "|" + b.getIsbn();
    } else if (item instanceof DVD d) {
        return "DVD|" + d.getTitle() + "|" + d.getIsbn() + "|" +
d.getRuntimeMinutes();
    }
    throw new IllegalStateException("Unknown LibraryItem subtype: " +
item.getClass());
}
```

Polimorfisme membantu di memori; rekonstruksi butuh tag

Setelah Book, DVD, dan ReferenceBook ada sebagai objek, memanggil describe() pada ketiganya secara polimorfik tidak membutuhkan informasi tipe sama sekali. Membangun ulang ketiga objek yang sama dari berkas teks datar adalah masalah yang sama sekali berbeda -- format berkas itu sendiri harus membawa tag yang menyatakan konstruktor mana yang dipanggil, karena teks polos tidak punya konsep "kelas" sama sekali.

BookNotFoundException milik Bab 6 diganti nama menjadi ItemNotFoundException di bab ini, dengan alasan yang sama seperti method Library sendiri diganti nama -- nama sebuah exception seharusnya mendeskripsikan apa yang sesungguhnya dilaporkannya, dan "BookNotFoundException" berhenti akurat sejak Library berhenti bersifat khusus-Book.

9.7 Polimorfisme dalam Aksi: Program Driver

Program driver membangun Library yang menampung satu Book, satu DVD, dan satu ReferenceBook, lalu menjalankan setiap ide yang diperkenalkan bab ini secara berurutan: borrow() polimorfik (di mana override ReferenceBook diam-diam menolak), laporan polimorfik, equals()/hashCode() berbasis nilai (objek Book kedua yang berbagi ISBN dengan yang pertama), narrowing instanceof untuk satu operasi yang sungguh khusus-DVD, dan round trip save/load penuh lewat format berkas bertag-tipe.

LibraryDemo.java (dipersingkat)

```
Library library = new Library();
library.addItem(new Book("Clean Code", "Robert C. Martin", "9780132350884",
2008));
library.addItem(new DVD("The Imitation Game", "9900000000001", 114));
library.addItem(new ReferenceBook("Encyclopedia of Computer Science", "Ralston &
Reilly", "9780123456789", 2003));

LibraryItem book = library.findByTitle("Clean Code");
LibraryItem dvd = library.findByTitle("The Imitation Game");
LibraryItem refBook = library.findByTitle("Encyclopedia of Computer Science");

Book copyOfSameBook = new Book("Clean Code (2nd copy)", "Robert C. Martin",
"9780132350884", 2008);
System.out.println(book.equals(copyOfSameBook)); // true -- ISBN sama, objek
berbeda

for (LibraryItem item : new LibraryItem[] { book, dvd, refBook }) {
    if (item instanceof DVD d) {
        System.out.println(d.getRuntimeMinutes() + " minutes");
    }
}
```

9.8 Mengkompilasi dan Menjalankan Program Anda

javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java ItemNotFoundException.java Library.java LibraryDemo.java, lalu java LibraryDemo. Tidak ada import baru yang dibutuhkan selain yang sudah dipakai Bab 6.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Library.java LibraryDemo.java
$ java LibraryDemo
```

9.9 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Backend Profitina secara rutin menemukan merek atau entitas dunia-nyata yang sama disebut lintas berbagai sumber berbeda -- URL berbeda, kata-kata persis berbeda -- dan perlu mengenali bahwa dua rekaman merujuk pada hal yang SAMA, bukan memperlakukannya sebagai tidak berkaitan. Itu persis masalah equals()/hashCode() bab ini dalam skala lebih besar: identitas (apakah ini objek literal yang sama di memori, atau baris database yang sama) adalah pertanyaan yang salah; kesetaraan nilai terhadap sebuah identifier kanonis (nama merek atau domain yang dinormalisasi, memainkan peran yang sama seperti isbn di bab ini) adalah yang benar, dan salah menerapkan hashCode() dengan cara

diam-diam yang sama seperti diperingatkan bab ini berarti entri duplikat diam-diam bertahan dari deduplikasi yang seharusnya menangkapnya.

9.10 Ringkasan Bab dan Poin-Poin Utama

- Pemanggilan method melakukan dispatch berdasarkan tipe runtime (dinamis) objek, tidak pernah tipe deklarasi (statis) -- ini adalah mekanisme lengkap di balik setiap loop polimorfik sejak Bab 7.
- Upcasting (subtipe ke supertipe) otomatis dan selalu aman; downcasting (supertipe ke subtipe) butuh cast eksplisit atau pattern-match instanceof, karena dugaannya bisa salah.
- Meng-override method tidak pernah mengharuskan pemanggilan super -- ReferenceBook.borrow() sengaja menggantikan perilaku warisan LibraryItem alih-alih memperluasnya.
- Pewarisan bisa berantai lebih dari satu tingkat (LibraryItem → Book → ReferenceBook); field protected yang dideklarasikan di puncak tetap terjangkau di dasar.
- Setiap kelas secara implisit extends Object dan mewarisi equals()/hashCode() berbasis identitasnya; meng-override keduanya bersama (tidak pernah hanya satu) mengalihkan kelas ke kesetaraan berbasis nilai, dibangun dari field yang sama di kedua method.
- Library kini menampung List<LibraryItem>, bukan List<Book> -- setiap method bersifat polimorfik kecuali persistensi berkas, yang membutuhkan tag tipe eksplisit untuk merekonstruksi objek konkret dari teks polos.

Sebuah hierarki hanya sungguh teruji pada hari seseorang memintanya melakukan satu hal yang tidak pernah dirancang untuknya -- dan hari Library akhirnya harus menampung DVD dan ReferenceBook berdampingan dengan Book, tanpa satu pun pemanggilnya menyadari perbedaannya, adalah hari itu.

9.11 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. ReferenceBook.borrow() mengembalikan false tanpa syarat dan tidak pernah memanggil super.borrow(). Tuliskan ulang agar method itu MEMANGGIL super.borrow() lebih dulu, dan jelaskan dalam satu kalimat mengapa perilaku hasilnya akan salah untuk item khusus-referensi.
2. Andaikan LibraryItem.equals() membandingkan title, bukan isbn. Berikan satu pasangan konkret objek Book yang akan menjadi "setara" secara salah di bawah perubahan itu, dan jelaskan mengapa isbn adalah pilihan yang lebih aman.
3. Tambahkan tingkat keempat ke hierarki: RareBook extends ReferenceBook, menambahkan field estimatedValue. Method ReferenceBook mana, jika ada, yang perlu di-override RareBook untuk mengubah perilaku, versus cukup diwarisi tanpa perubahan?
4. Library.toLine() memeriksa `instanceof ReferenceBook` sebelum `instanceof Book`. Jelaskan apa yang akan salah jika urutan itu dibalik, mengingat setiap ReferenceBook juga sebuah Book.
5. Dengan kata-kata Anda sendiri, jelaskan perbedaan antara ClassCastException yang dilempar oleh cast (LibraryItem) item yang tidak diperiksa, dan bentuk pattern-match instanceof yang

digunakan di seluruh bab ini -- yang mana yang sungguh bisa gagal saat runtime, dan yang mana yang tidak bisa?

Lampiran 9.A — Log Verifikasi Eksekusi

Berkas lengkap LibraryItem.java, Book.java, DVD.java, ReferenceBook.java, Renewable.java, ItemNotFoundException.java, Library.java, dan LibraryDemo.java (Code/Java/Chapter09/, disertakan bersama buku ini) dikompilasi dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kodenya benar, bukan sekadar "terlihat benar". Output direproduksi verbatim di bawah.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Library.java LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
--- Borrow attempts (polymorphic dispatch) ---
Clean Code .borrow() -> true
The Imitation Game .borrow() -> true
Encyclopedia .borrow() -> false  (ReferenceBook overrides borrow() to always
refuse -- never calls super.borrow())

--- equals()/hashCode(): value equality, not identity ---
book == copyOfSameBook       -> false  (different objects in memory)
book.equals(copyOfSameBook)  -> true  (same ISBN -> same real-world item)
book.hashCode() == copy's     -> true

--- instanceof pattern matching: the one DVD-only operation ---
Clean Code is not a DVD -- no runtime to report
The Imitation Game runtime: 114 minutes
Encyclopedia of Computer Science is not a DVD -- no runtime to report

--- removeItemOrThrow on a missing ISBN ---
Caught: No library item found with ISBN: 0000000000000
```

Referensi

[1] Oracle Corporation. "Overriding and Hiding Methods." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/override.html (diakses Agustus 2026).

[2] Oracle Corporation. "Polymorphism." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html (diakses Agustus 2026).

[3] Oracle Corporation. "Object." Java SE 21 API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html (diakses Agustus 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Item 10 ("Obey the general contract when overriding equals") dan Item 11 ("Always override hashCode when you override equals").

BAB 10

Generik & Kerangka Kerja Koleksi

Bab 9 akhirnya membuat Library mampu menampung LibraryItem apa pun, tetapi meninggalkannya dengan tepat satu pola akses (pemindaian linear berdasarkan title) dan tanpa urutan terjamin ketika campuran Book, DVD, dan ReferenceBook. Bab ini menambahkan struktur inti lain milik Collections Framework -- indeks Map berdampingan dengan List -- plus kelas generik Pair yang bisa dipakai ulang, method utilitas berwildcard-terbatas, urutan alami berbasis Comparable, dan Set berisi tipe konkret yang berbeda-beda.

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan apa yang direpresentasikan parameter tipe sebuah kelas generik (Pair<A, B>), dan mengapa type erasure membuat Pair<String, Integer> dan Pair<LibraryItem, Double> berbagi tepat satu berkas .class saat runtime. (2) Membaca dan menulis parameter bertipe wildcard terbatas (List<? extends LibraryItem>), dan menjelaskan mengapa List<Book> bisa dioper ke tempat yang mengharapkannya meskipun List<Book> sendiri bukan List<LibraryItem>. (3) Menambahkan indeks kedua berbasis Map ke kelas koleksi berbasis List yang sudah ada tanpa merusak method mana pun yang sudah ada, menjaga kedua struktur tetap diperbarui bersama pada setiap penyisipan dan penghapusan. (4) Mengimplementasikan Comparable agar Collections.sort() menghasilkan urutan alami tanpa Comparator terpisah, dan menjelaskan dari mana urutan itu diturunkan. (5) Menjelaskan perbedaan antara Set yang menjaga urutan insersi (LinkedHashSet) dan yang tidak menjamin urutan apa pun (HashSet), serta menyatakan yang mana yang dibutuhkan sebuah log yang harus reproducible.

10.1 Rekap: Pustaka Sejauh Ini

Bab 9 menggeneralisasi field milik Library dari List<Book> menjadi List<LibraryItem> dan memformalkan tipe statis vs. tipe runtime, tetapi setiap method Library masih melakukan tepat satu hal terhadap List itu: mencarinya secara linear berdasarkan title. Masih belum ada cara mencari item berdasarkan ISBN lebih cepat daripada O(n), tidak ada urutan terjamin ketika sebuah laporan mencampur Book, DVD, dan ReferenceBook, dan tidak ada cara mengetahui, sekilas saja, subtipe konkret mana yang sungguh dimiliki sebuah Library. Bentuk bab ini menjawab ketiganya, menggunakan dua pilar lain Java Collections Framework yang belum pernah dibutuhkan mata kuliah ini: Map dan Set.

Gambar 10.1 — Peta jalan 16-kuliah OOP. Bab ini adalah Kuliah 10: kelas generik, method berwildcard-terbatas, dan indeks kedua berbasis Map/Set berdampingan dengan List yang sudah dimiliki Library.

10.2 Kelas Generik yang Bisa Dipakai Ulang: Pair<A, B>

Memasangkan sebuah LibraryItem dengan nilai hasil hitungan -- jumlah dendanya, misalnya -- atau sebuah title dengan tahun terbit, adalah bentuk yang berulang di sepanjang program, dan menulis kelas khusus untuk setiap kombinasi tipe berarti menulis empat baris boilerplate yang sama berulang-ulang. Pair<A, B> menulis bentuk itu tepat sekali: A dan B adalah parameter tipe, bukan tipe sungguhan -- compiler mensubstitusi tipe sungguhan di setiap titik pemanggilan (Pair<LibraryItem, Double>, Pair<String, Integer>, ...) dan memeriksa bahwa setiap penggunaan Pair tersebut konsisten dengan tipe yang dipilih di sana (Gambar 10.1 dan 10.2).

Pair.java (lengkap)

```
public final class Pair<A, B> {
    private final A first;
    private final B second;

    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }

    public A getFirst() { return first; }
    public B getSecond() { return second; }

    @Override
    public String toString() {
        return "(" + first + ", " + second + ")";
    }
}
```

Generik, Dibandingkan: Diperiksa Saat Compile vs. Tidak Pernah Diperiksa

Kedua bahasa membiarkan satu kelas/fungsi bekerja untuk banyak tipe -- tetapi apa yang sungguh terjadi pada informasi tipe saat runtime sangat berbeda.

	Java: List<T>, Pair<A,B>, <T extends X>	Python: list[T], Pair[A, B], T: X
Diperiksa kapan?	Saat COMPILE -- javac menolak pemanggilan yang tidak cocok sebelum pernah dijalankan	TIDAK PERNAH oleh interpreter itu sendiri -- hanya oleh alat opsional (mypy, pyright) yang Anda jalankan terpisah
Apa yang bertahan sampai runtime?	Tidak ada -- type erasure menghapus seluruh info generik dari berkas .class; List<String> dan List<Integer> berbagi SATU objek kelas saat runtime	Tidak ada yang ditegakkan juga, tapi alasannya beda -- CPython tidak pernah membaca anotasi tipe sebagai aturan, hanya sebagai metadata opsional
Hasil praktis	Ketidakcocokan tipe adalah error COMPILE, tertangkap sebelum program pernah berjalan	Ketidakcocokan tipe tidak terlihat sampai menyebabkan bug nyata -- atau sampai static checker dijalankan dan melaporkannya

Tak satu pun bahasa memeriksa generik SAAT PROGRAM BERJALAN -- Java memeriksa sekali, saat compile, lalu membuang informasinya; Python tidak pernah memeriksa sama sekali kecuali alat terpisah diminta. "Generik Java dihapus (erased)" dan "generik Python tidak ditegakkan" adalah klaim yang berbeda.

Gambar 10.2 — Generik dibandingkan: diperiksa sekali saat compile lalu dihapus (Java) vs. tidak pernah diperiksa sama sekali oleh interpreter (Python).

Type erasure: satu kelas, bukan satu per kombinasi tipe

Setelah javac memeriksa setiap penggunaan Pair<LibraryItem, Double> untuk konsistensi, javac membuang informasi generiknya -- Pair.class yang terkompilasi menyimpan field Object polos, tanpa jejak A atau B tersisa di dalamnya. Sebuah Pair<String, Integer> dan sebuah Pair<LibraryItem, Double>, saat runtime, secara literal adalah kelas yang sama; pair.getClass() tidak pernah bisa memberi tahu Anda apa A atau B sebenarnya. Pemeriksaannya tetap terjadi -- hanya sekali, saat compile, bukan pada setiap pemanggilan method seperti pemeriksaan if/instanceof.

10.3 Wildcard Terbatas: List<? extends LibraryItem>

LibraryUtils.totalFines() perlu menjumlahkan calculateFine() lintas sebuah List item, tetapi seharusnya tidak perlu peduli apakah List itu menampung LibraryItem, hanya Book, atau hanya DVD -- ketiganya adalah argumen yang masuk akal, karena yang dilakukannya hanya membaca setiap item dan memanggil method yang dijamin setiap LibraryItem. Menulis parameter sebagai List<LibraryItem> polos akan menolak List<Book> secara langsung: generik Java bersifat invarian, sehingga List<Book> BUKAN List<LibraryItem>, meskipun Book ADALAH sebuah LibraryItem. Wildcard terbatas List<? extends LibraryItem> adalah persis celah yang ditutup ini -- ia menerima List berisi LibraryItem atau subtipe apa pun, dengan syarat hanya bisa dipakai untuk MEMBACA item keluar (menambahkan ke dalamnya sengaja dibatasi, karena compiler tidak bisa tahu persis subtipe mana yang perlu diterimanya).

LibraryUtils.java (lengkap)

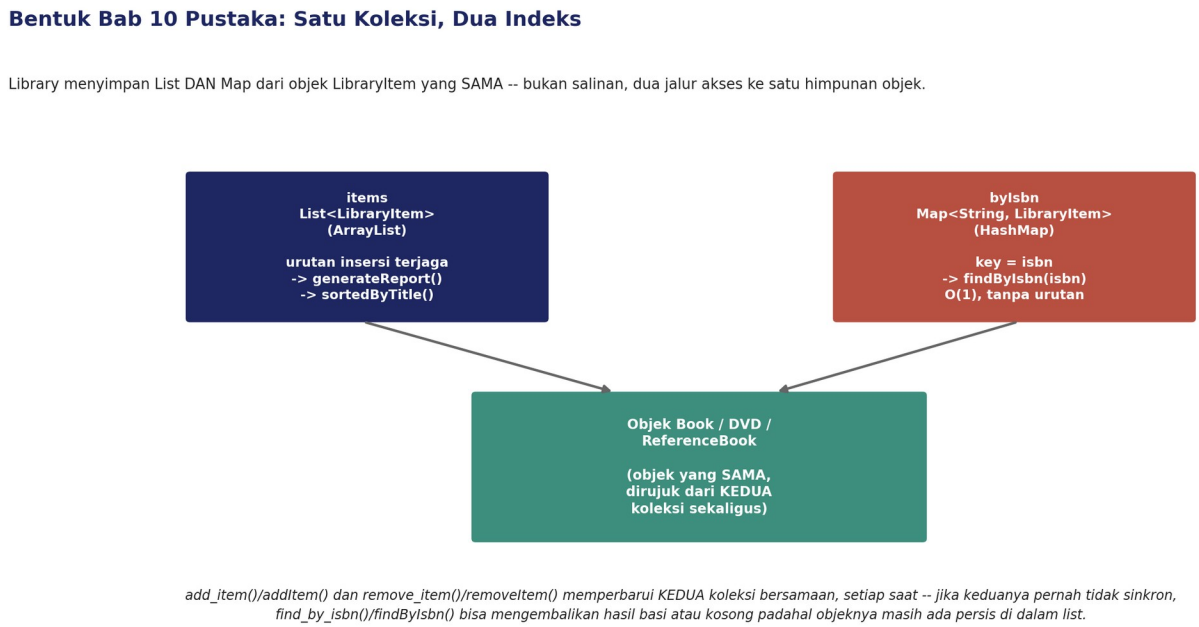
```
public final class LibraryUtils {

    private LibraryUtils() { } // kelas utilitas: tidak pernah diinstansiasi

    public static double totalFines(List<? extends LibraryItem> items, int
daysOverdue) {
        double total = 0.0;
        for (LibraryItem item : items) {
            total += item.calculateFine(daysOverdue);
        }
        return total;
    }
}
```

Wildcard terbatas bukan hal yang sama dengan method generik

totalFines() BUKAN method generik -- signature-nya tidak mendeklarasikan parameter tipe miliknya sendiri (tidak ada <T> sebelum tipe kembalian), hanya wildcard di dalam tipe sebuah parameter biasa. Method generik sungguhan di sini akan berbunyi public static <T extends LibraryItem> double totalFines(List<T> items, int daysOverdue) -- kedua bentuk sama-sama terkompilasi dan bekerja untuk kasus penggunaan ini, tetapi keduanya menjawab pertanyaan berbeda (bentuk wildcard berkata "saya hanya perlu membaca LibraryItem, saya tidak peduli T sebenarnya apa," sementara bentuk parameter-tipe akan membiarkan tubuh method juga merujuk T dengan namanya). Jalur Python pada Bagian 10.3 buku itu menggunakan bentuk parameter-tipe secara langsung, karena sintaks fungsi-generik Python yang lebih baru membuat itu pilihan yang lebih alami di sana -- lihat kotak callout pada bagian tersebut.

10.4 Satu Koleksi, Dua Indeks: byIsbn Berdampingan dengan items

Field items milik Library (sebuah List<LibraryItem>, tidak berubah sejak Bab 9) di bab ini bergabung dengan field kedua, byIsbn, sebuah Map<String, LibraryItem> berkunci ISBN -- addItem dan removeItem kini memperbarui kedua koleksi bersamaan, setiap saat, sehingga keduanya tidak pernah tidak-sinkron. findByIsbn(isbn) kemudian menjawab dalam O(1) apa yang masih dijawab findByTitle() dalam O(n): keduanya mencari objek LibraryItem yang persis sama, hanya lewat dua jalur berbeda (Gambar 10.3).



Gambar 10.3 — Bentuk Bab 10 Pustaka: indeks List dan indeks Map milik Library, merujuk objek LibraryItem yang sama.

Library.java -- indeks byIsbn (dipersingkat)

```
public class Library {
    private List<LibraryItem> items;
    private Map<String, LibraryItem> byIsbn;

    public Library() {
        this.items = new ArrayList<>();
        this.byIsbn = new HashMap<>();
    }

    public void addItem(LibraryItem item) {
        items.add(item);
        byIsbn.put(item.getIsbn(), item);
    }

    public boolean removeItem(String isbn) {
        for (int i = 0; i < items.size(); i++) {
            if (items.get(i).getIsbn().equals(isbn)) {
                items.remove(i);
                byIsbn.remove(isbn);
                return true;
            }
        }
        return false;
    }

    public LibraryItem findByIsbn(String isbn) {
        return byIsbn.get(isbn);
    }
}
```

Mengapa tetap mempertahankan List, kalau sudah ada Map?

Sebuah HashMap saja sudah cukup menjawab findByIsbn() dengan baik, tetapi ia sama sekali tidak menjanjikan urutan iterasi apa pun -- generateReport() dan sortedByTitle() bab ini sama-sama membutuhkan titik awal yang stabil dan terurut-insersi untuk bekerja. Library mempertahankan kedua struktur karena keduanya melayani pekerjaan yang sungguh berbeda: List adalah sumber kebenaran untuk urutan, Map adalah indeks kedua yang buta-urutan, dibangun murni demi kecepatan lookup O(1).

10.5 Urutan Alami: LibraryItem implements Comparable<LibraryItem>

Collections.sort(list) (atau list.sort(null)) perlu tahu cara mengurutkan dua LibraryItem, dan ia mendapatkannya baik dari argumen Comparator eksplisit maupun dari tipe elemen itu sendiri yang mengimplementasikan Comparable -- Bab 10 memilih yang kedua, sehingga setiap List<LibraryItem> dalam program ini memiliki satu urutan default yang jelas (berdasarkan title) tanpa pemanggil mana pun perlu menyediakan Comparator hanya untuk mengurutkan sebuah list.

LibraryItem.java -- compareTo() (baru di bab ini)

```
public abstract class LibraryItem implements Comparable<LibraryItem> {
    // ... title, isbn, onLoan, equals()/hashCode() tidak berubah dari Bab 9 ...

    @Override
    public int compareTo(LibraryItem other) {
        return this.title.compareToIgnoreCase(other.title);
    }
}
```

Library.java -- sortedByTitle()

```
public List<LibraryItem> sortedByTitle() {
    List<LibraryItem> copy = new ArrayList<>(items);
    Collections.sort(copy);
    return copy;
}
```

Urutan alami vs. Comparator eksplisit

Comparable.compareTo() menyediakan urutan ALAMI sebuah kelas -- satu urutan default yang diklaim masuk akal oleh kelas itu sendiri, di sini berdasarkan title, tidak peduli huruf besar/kecil. Pemanggil yang menginginkan urutan berbeda (berdasarkan ISBN, jumlah denda, tahun terbit) tidak pernah terjebak pada title: mengoper Comparator eksplisit ke Collections.sort(list, comparator) menimpa urutan alami untuk satu pemanggilan itu saja, tanpa mengubah compareTo() atau memengaruhi pemanggil lain mana pun.

10.6 LinkedHashSet Berisi Tipe-Tipe yang Berbeda

distinctTypes() menjawab pertanyaan yang tidak bisa dijawab Library dalam satu baris sebelum bab ini: subtipe konkret mana yang sungguh dimiliki Library ini sekarang? Seluruh kontrak sebuah Set adalah "tanpa duplikat" -- tiga Book menyumbang string "Book" ke hasil hanya sekali, berapa pun banyaknya objek Book yang sungguh ada -- dan Library sengaja membangun Set itu sebagai LinkedHashSet, bukan HashSet polos, khusus agar hasilnya kembali dalam urutan pertama-kali-terlihat setiap saat, dari satu eksekusi ke eksekusi lain.

Library.java -- distinctTypes()

```
public Set<String> distinctTypes() {
    Set<String> types = new LinkedHashSet<>();
    for (LibraryItem item : items) {
        types.add(item.getClass().getSimpleName());
    }
    return types;
}
```

HashSet polos tidak akan merusak kodenya -- hanya lognya

Menukar LinkedHashSet dengan HashSet polos di sini akan tetap menghasilkan Set<String> yang benar dan bebas-duplikat; tidak ada apa pun pada KEBENARAN method ini yang bergantung pada implementasi Set mana yang dipakai. Yang tidak dijamin HashSet polos adalah urutan mana ketiga nama tipe itu tercetak ketika Set-nya di-iterasi -- dan log eksekusi Lampiran 10.A membutuhkan urutan itu tetap sama pada setiap eksekusi agar transkrip di bawah bisa dipercaya sebagai REPRODUKSI, bukan cuplikan satu kali. Kebutuhan reproduktibilitas itu, bukan kebenaran, adalah persis alasan LinkedHashSet dipilih di sini.

10.7 Menyatukan Semuanya: Program Driver

Driver membangun Library yang menampung empat item -- trio Bab 9 plus Book keempat, Algorithms karya Sedgewick -- lalu menjalankan lima tambahan bab ini secara berurutan: lookup findByIsbn() O(1), sortedByTitle() yang langsung menjadi masukan totalFines() berwildcard-terbatas, distinctTypes() (kutipan Bagian 10.6 sudah menunjukkan yang satu ini), dan akhirnya dua instance Pair<A, B> yang dibangun dari data yang sama. Kutipan dipersingkat di bawah menunjukkan yang pertama, ketiga, dan kelima secara langsung; urutan lengkapnya, plus langkah rekap Bab 9 yang mendahuluinya, ada di Code/Java/Chapter10/LibraryDemo.java.

LibraryDemo.java (dipersingkat)

```
Library library = new Library();
library.addItem(new Book("Clean Code", "Robert C. Martin", "9780132350884",
2008));
library.addItem(new DVD("The Imitation Game", "9900000000001", 114));
library.addItem(new ReferenceBook("Encyclopedia of Computer Science", "Ralston &
Reilly", "9780123456789", 2003));
library.addItem(new Book("Algorithms", "Robert Sedgewick", "9780321573513",
2011));

LibraryItem foundByIsbn = library.findByIsbn("9780321573513");
System.out.println(foundByIsbn.getTitle());

List<LibraryItem> byTitle = library.sortedByTitle();
double allFines = LibraryUtils.totalFines(byTitle, 10);
System.out.printf("$%.2f%n", allFines);

LibraryItem book = library.findByTitle("Clean Code");
Pair<LibraryItem, Double> mostExpensive = new Pair<>(book,
book.calculateFine(10));
System.out.println(mostExpensive);
Pair<String, Integer> titleAndYear = new Pair<>("Clean Code", 2008);
System.out.println(titleAndYear);
```

10.8 Mengkompilasi dan Menjalankan Program Anda

javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java ItemNotFoundException.java Pair.java LibraryUtils.java Library.java LibraryDemo.java, lalu java LibraryDemo. Tidak ada import baru yang dibutuhkan selain java.util.Map, java.util.HashMap,

java.util.LinkedHashSet, dan java.util.Set, semuanya bagian dari paket java.util yang sama tempat Bab 5 sudah meng-import ArrayList dan List.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Pair.java LibraryUtils.java Library.java
LibraryDemo.java
$ java LibraryDemo
```

10.9 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Backend Profitina menyimpan list terurut berisi sebutan merek yang dilacak untuk laporan peringkatnya, dan, berdampingan dengannya, sebuah map ber-hash-indeks berkunci identifier eksternal kanonis untuk lookup deduplikasi O(1) ketika sebutan baru datang -- persis bentuk items/byIsbn bab ini, dengan alasan yang persis sama: satu struktur menjaga urutan yang dibutuhkan laporan, yang lain menukar urutan demi kecepatan lookup. Utilitas pemasangan generik bersama (Pair bab ini, dalam semangatnya) muncul di mana pun backend perlu membawa skor hasil hitungan berdampingan dengan rekaman tempat skor itu dihitung, tanpa menulis kelas dua-field khusus untuk setiap jenis skor yang ditambahkannya.

10.10 Ringkasan Bab dan Poin-Poin Utama

- Parameter tipe generik (Pair<A, B>) diperiksa untuk konsistensi sekali, saat compile, lalu dihapus -- saat runtime hanya ada tepat satu Pair.class, dipakai bersama oleh setiap kombinasi A/B yang pernah digunakan.
- Wildcard terbatas (List<? extends LibraryItem>) bukan method generik -- ia memungkinkan method biasa menerima List<LibraryItem> atau List<subtipe> apa pun, dengan syarat hanya membaca dari list itu, karena generik Java bersifat invarian dan sebaliknya akan menolak List<Book> secara langsung.
- Library kini mempertahankan Map<String, LibraryItem> (byIsbn) berdampingan dengan List<LibraryItem> (items), diperbarui bersama pada setiap penyisipan dan penghapusan, menukar ketiadaan urutan pada Map demi lookup ISBN O(1).
- Mengimplementasikan Comparable memberi kelas urutan alami yang dipakai Collections.sort() tanpa argumen Comparator; Comparator eksplisit tetap bisa menimpa default itu untuk satu pemanggilan, tanpa menyentuh compareTo().
- LinkedHashSet, bukan HashSet polos, adalah yang membuat urutan keluaran distinctTypes() reproducible lintas eksekusi -- keduanya sama-sama benar sebagai Set, tetapi hanya satu yang menjamin urutan iterasi yang bisa diandalkan sebuah log verifikasi.

Sebuah kelas koleksi baru layak disebut "framework" pada hari ia berhenti menjadi sekadar List dengan method tambahan yang ditempel -- dan hari Library tumbuh Map yang harus tetap sinkron

dengan List-nya, pada setiap penyisipan dan penghapusan, tanpa satu pun boleh tertinggal, adalah hari itu.

10.11 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. LibraryUtils.totalFines() menerima List<? extends LibraryItem>, bukan List<LibraryItem>. Berikan satu titik pemanggilan dari LibraryDemo.java yang akan gagal dikompilasi jika parameternya diubah menjadi List<LibraryItem> polos, dan jelaskan mengapa.
2. Andaikan Library.removeItem(isbn) lupa memanggil byIsbn.remove(isbn), sehingga hanya menghapus item dari items. Deskripsikan satu urutan pemanggilan konkret setelah titik itu yang akan mengembalikan jawaban salah (tetapi tidak jelas-jelas salah).
3. Tuliskan ulang LibraryUtils.totalFines() sebagai method generik sungguhan dengan parameter tipe terbatas (public static <T extends LibraryItem> double totalFines(List<T> items, int daysOverdue)) alih-alih wildcard. Apakah setiap titik pemanggilan yang ada di LibraryDemo.java akan tetap terkompilasi tanpa perubahan? Jelaskan jawaban Anda.
4. LibraryItem.compareTo() mengurutkan berdasarkan title, tidak peduli huruf besar/kecil. Tuliskan Comparator<LibraryItem> alternatif yang mengurutkan berdasarkan calculateFine(10) secara menurun, dan berikan satu baris kode yang akan memakainya tanpa mengubah compareTo() sama sekali.
5. Jelaskan, dengan kata-kata Anda sendiri, mengapa Pair.toString() bisa dengan aman menulis "(" + first + ", " + second + ")" tanpa perlu tahu, baik saat compile maupun saat runtime, tipe konkret apa yang ternyata dipakai A dan B.

Lampiran 10.A — Log Verifikasi Eksekusi

Berkas lengkap LibraryItem.java, Renewable.java, Book.java, DVD.java, ReferenceBook.java, ItemNotFoundException.java, Pair.java, LibraryUtils.java, Library.java, dan LibraryDemo.java (Code/Java/Chapter10/, disertakan bersama buku ini) dikompilasi dan dijalankan pada OpenJDK 21.0.10 sebelum bab ini ditulis, untuk memastikan kodenya benar, bukan sekadar "terlihat benar". Output direproduksi verbatim di bawah.

```
$ javac LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
      ItemNotFoundException.java Pair.java LibraryUtils.java Library.java
LibraryDemo.java
(tidak ada error)

$ java LibraryDemo
--- Borrow attempts (polymorphic dispatch, Chapter 9) ---
Clean Code .borrow() -> true
The Imitation Game .borrow() -> true
Encyclopedia .borrow() -> false  (ReferenceBook overrides borrow() to always
refuse -- never calls super.borrow())

--- equals()/hashCode(): value equality, not identity (Chapter 9) ---
book.equals(copyOfSameBook)  -> true  (same ISBN -> same real-world item)

--- NEW Chapter 10: findByIsbn(), the Map-backed O(1) lookup ---
findByIsbn("9780321573513") -> Algorithms

--- NEW Chapter 10: sortedByTitle(), via LibraryItem implements Comparable ---
  Algorithms
  Clean Code
  Encyclopedia of Computer Science
  The Imitation Game

--- NEW Chapter 10: distinctTypes(), a Set<String> (no duplicates) ---
distinctTypes() -> [Book, DVD, ReferenceBook]

--- NEW Chapter 10: LibraryUtils.totalFines(), using a bounded wildcard (List<?
extends LibraryItem>) ---
totalFines(all items, 10 days overdue) -> $20.50

--- NEW Chapter 10: Pair<A, B>, a generic class ---
Pair<LibraryItem, Double> -> ("Clean Code" by Robert C. Martin [ISBN
9780132350884, 2008] - ON LOAN (renewed 0x), 6.5)
Pair<String, Integer>     -> (Clean Code, 2008)

--- removeItemOrThrow on a missing ISBN (Chapter 6/9) ---
Caught: No library item found with ISBN: 0000000000000

--- Save/load round trip (type-tagged file format, Chapter 6/9) ---
Reloaded 4 item(s), distinct types: [Book, DVD, ReferenceBook]
```

Referensi

[1] Oracle Corporation. "Generics (Updated)." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/generics/index.html (diakses Agustus 2026).

[2] Oracle Corporation. "Type Erasure." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/generics/erasure.html (diakses Agustus 2026).

[3] Oracle Corporation. "LinkedHashSet<E>." Java SE 21 API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedHashSet.html (diakses Agustus 2026).

[4] J. Bloch. Effective Java, 3rd ed. Addison-Wesley, 2017, Item 26 ("Don't use raw types") dan Item 31 ("Use bounded wildcards to increase API flexibility").

BAB 11

Pemrograman GUI: Event & MVC

Setiap bab sebelumnya, Pustaka selalu berupa program konsol: sebuah method driver yang berjalan dari atas ke bawah sekali, mencetak hasilnya, lalu berhenti. Bab ini memberi Pustaka antarmuka berjendela yang nyata -- daftar item, formulir untuk menambah buku, tombol untuk meminjam/mengembalikan/menghapus -- dibangun dengan JavaFX dan diorganisasi dengan pola Model-View-Controller (MVC), yang memecah program menjadi Model yang tidak tahu apa-apa tentang GUI, View yang tidak tahu apa-apa tentang logika perpustakaan, dan Controller yang menjadi satu-satunya kelas yang boleh menyentuh keduanya (Gambar 11.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

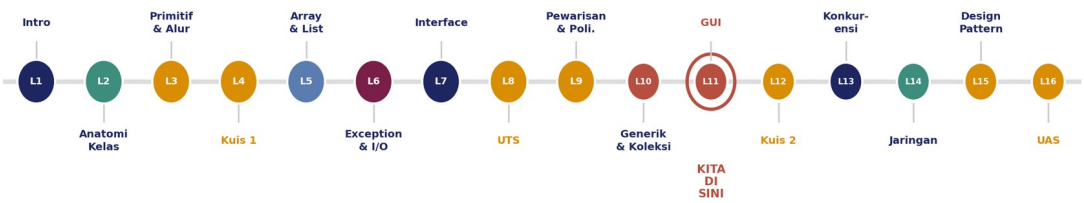
(1) Menjelaskan apa yang dipisahkan oleh pola Model-View-Controller, dan menyatakan kelas mana boleh bergantung pada kelas mana. (2) Membaca dan menulis event handler JavaFX yang dipasang dengan setOnAction(...), dan menjelaskan kapan JavaFX sungguh memanggilnya. (3) Membangun kelas View JavaFX yang tugasnya semata-mata membangun dan menyediakan kontrol, tanpa logika bisnis sama sekali. (4) Menelusuri satu klik tombol dari awal sampai akhir: View mengekspos klik -> handler Controller membaca View -> Controller mengubah Model -> Controller menyuruh View menyegarkan tampilannya. (5) Menjelaskan mengapa eksekusi verifikasi terskrip tanpa layar (memilih baris dan memicu tombol secara terprogram) membuktikan jalur kode yang sama persis dengan klik mouse sungguhan, dan bukan simulasi GUI.

11.1 Rekap: Pustaka Sejauh Ini

Bab 10 memberi Library indeks kedua (Map<String, LibraryItem> berkunci ISBN, berdampingan dengan List<LibraryItem>-nya), kelas generik Pair<A, B> yang bisa dipakai ulang, method utilitas berwildcard-terbatas, dan urutan alami lewat Comparable -- tetapi setiap bab hingga Bab 10 masih berinteraksi dengan Library itu tidak lebih dari sekadar rangkaian pemanggilan System.out.println() yang dipilih sekali, saat compile, oleh programmer. Tidak ada apa pun di LibraryDemo.java yang membiarkan seseorang memilih item mana yang mau dipinjam selagi program berjalan; driver sudah memutuskan semuanya lebih dulu. Bentuk bab ini menjawab itu: sebuah jendela nyata, tempat seseorang mengklik baris list sungguhan dan tombol sungguhan, dan program merespons apa pun yang diklik orang itu, bukan skrip yang ditulis sebelumnya.

Peta Jalan 16 Kuliah OOP

Bab ini adalah Kuliah 11: membangun GUI berjendela nyata di atas Model yang sudah dibangun mata kuliah ini, lewat pola MVC.



Gambar 11.1 — Peta jalan 16-kuliah OOP. Bab ini adalah Kuliah 11: GUI berjendela nyata dibangun di atas Model Library/LibraryItem yang sama, yang sudah dibangun mata kuliah ini sejak Bab 5, lewat pola Model-View-Controller.

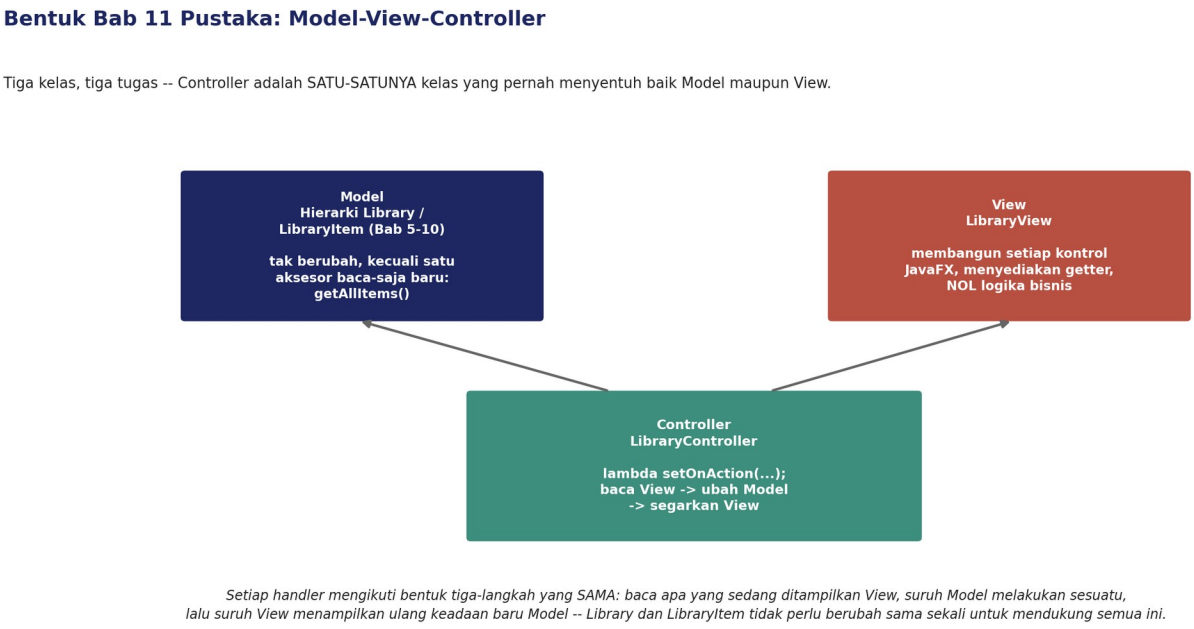
11.2 Dari Konsol ke Jendela: Pemrograman Berbasis Event

Sebuah driver konsol seperti LibraryDemo.java adalah program yang MEMEGANG KENDALI: ia memutuskan, baris demi baris, apa yang terjadi selanjutnya, dan memanggil System.in untuk input (jika pernah dilakukannya) hanya akan menjeda urutan tetap yang sama, tidak pernah mengubah bentuknya. Program GUI membalik hubungan itu sepenuhnya -- begitu LibraryApp.start() selesai membangun jendela dan kembali, kode program itu sendiri berhenti mengendalikan apa pun. Event loop milik JavaFX mengambil alih, duduk diam sampai pengguna melakukan sesuatu (mengklik tombol, mengetik di field, menutup jendela), lalu memanggil balik tepat satu potongan kecil kode milik programmer untuk setiap event semacam itu. Gaya ini disebut PEMROGRAMAN BERBASIS EVENT: logika program adalah kumpulan callback, masing-masing pendek, masing-masing dipicu oleh sesuatu yang dilakukan pengguna, tanpa method mana pun yang menggambarkan keseluruhan eksekusi dari awal sampai akhir seperti main() dulu.

> **Event loop tidak pernah terblokir menunggu kode ANDA**
>
> Event loop JavaFX memanggil sebuah handler, menunggunya selesai, lalu kembali menunggu event berikutnya -- ia tidak menjalankan handler Anda bersamaan dengan hal lain. Inilah persis alasan setiap handler di LibraryController bab ini pendek dan langsung selesai: handler yang berjalan lama (misalnya panggilan jaringan yang lambat) akan membekukan seluruh jendela selama durasi itu, karena tidak ada apa pun -- bahkan menggambar ulang jendela -- yang bisa terjadi selagi event loop berada di dalam callback Anda. (Materi konkurensi Bab 13 adalah persis alat untuk handler yang sungguh perlu melakukan pekerjaan lambat tanpa membekukan UI; semua handler bab ini cukup cepat sehingga pertanyaan itu tidak pernah muncul.)

11.3 Pola Model-View-Controller

MVC memecah program GUI menjadi tiga kelas dengan tiga tugas terpisah, dan tepat satu hubungan yang diizinkan di antara mereka: Controller boleh bergantung pada Model maupun View, tetapi Model tidak boleh pernah meng-import atau merujuk apa pun tentang View, dan View tidak boleh pernah memanggil apa pun pada Model secara langsung. Model milik Pustaka -- hierarki Library dan LibraryItem -- ditulis sepanjang Bab 5 hingga 10 tanpa memikirkan GUI sama sekali, dan bab ini hanya perlu menambahkan tepat satu method padanya (getAllItems(), aksesor salinan-defensif) agar bisa dipakai dari View. Setiap method Model lain yang sudah ditulis Bab 5-10 -- addItem(), removeItem(), findByTitle(), borrow(), returnItem() -- dipakai ulang sepenuhnya tanpa perubahan (Gambar 11.2).



Gambar 11.2 — Bentuk Bab 11 Pustaka: Model, View, dan Controller, dengan Controller sebagai satu-satunya kelas yang menyentuh keduanya.

Model-View-Controller, secara presisi

MODEL: data dan aturan tentangnya (Library, LibraryItem, Book, DVD, ReferenceBook) -- tidak tahu apa-apa tentang tombol, jendela, atau piksel. VIEW: kontrol yang terlihat dan tata letaknya (LibraryView) -- tidak tahu apa-apa tentang apa artinya meminjam atau menghapus sebuah item, hanya tahu cara menampilkan LibraryItem dan menyediakan kontrolnya sendiri. CONTROLLER: perekat (LibraryController) -- membaca apa yang sedang ditampilkan View, menyuruh Model melakukan sesuatu, lalu menyuruh View menampilkan ulang keadaan baru Model. Sebuah View yang dibangun terhadap Library yang sama sekali berbeda akan terlihat identik piksel demi piksel; sebuah Model yang dipakai oleh View yang sama sekali berbeda akan berperilaku identik.

11.4 View: LibraryView

LibraryView membangun setiap kontrol JavaFX yang terlihat di dalam konstruktornya -- sebuah ListView<LibraryItem> untuk item, empat TextField untuk menambah buku, empat tombol, dan sebuah Label status -- dan menyediakan masing-masing lewat getter publik. Ia tidak memanggil apa pun pada Library, dan tidak memiliki method yang memutuskan apa artinya meminjam, menambah, atau menghapus; satu-satunya method substantif selain getter adalah refresh(List<LibraryItem>), yang sekadar menggambar ulang daftar item apa pun yang diberikan kepadanya.

LibraryView.java (kutipan)

```
public class LibraryView {
    private final ListView<LibraryItem> itemList = new ListView<>();
    private final Button borrowButton = new Button("Borrow Selected");
    private final Button returnButton = new Button("Return Selected");
    private final Button removeButton = new Button("Remove Selected");
    private final Label statusLabel = new Label("Ready.");
    // ... titleField/authorField/isbnField/yearField, addButton ...

    public LibraryView() {
        itemList.setCellFactory(list -> new ListCell<>() {
            @Override
            protected void updateItem(LibraryItem item, boolean empty) {
                super.updateItem(item, empty);
                setText(empty || item == null ? null : item.describe());
            }
        });
        // ... membangun GridPane form, tata letak HBox/VBox,
root.setCenter()/setRight() ...
    }

    public ListView<LibraryItem> getItemList() { return itemList; }
    public Button getBorrowButton() { return borrowButton; }
    // ... satu getter per kontrol ...

    public void refresh(List<LibraryItem> items) {
        LibraryItem selected = itemList.getSelectionModel().getSelectedItem();
        itemList.getItems().setAll(items);
        if (selected != null && items.contains(selected)) {
            itemList.getSelectionModel().select(selected);
        }
    }

    public void setStatus(String message) { statusLabel.setText(message); }
}
```

Cell factory adalah cara sebuah ListView menampilkan objek kustom

Secara default, ListView<LibraryItem> JavaFX akan memanggil toString() pada setiap item untuk memutuskan teks apa yang ditampilkan -- yang akan bekerja, tetapi mengikat View pada toString() apa pun yang kebetulan dimiliki sebuah kelas Model. cellFactory di atas justru memanggil describe() secara eksplisit, method yang persis sama yang sudah diekspos setiap LibraryItem tepat untuk tujuan ini (Bab 7-9) -- View yang memilih cara menampilkan sebuah item, bukan diam-diam mewarisi apa pun yang kebetulan dihasilkan toString() milik Model.

11.5 Controller: Memasang Event

LibraryController adalah satu-satunya kelas di bab ini yang meng-import baik Library maupun LibraryView. Konstruktornya memanggil wireEvents() (memasang satu lambda setOnAction(...) per tombol) lalu refreshView() sekali, sehingga jendela terbuka sudah menampilkan keadaan awal Model. Setiap handler mengikuti bentuk tiga-langkah yang sama: baca seleksi atau field formulir View saat ini, suruh Model melakukan sesuatu, lalu panggil refreshView() agar View menampilkan ulang apa pun keadaan Model sekarang (Gambar 11.3).

Penanganan Event, Dibandingkan: Dua Toolkit, Satu Gagasan

Kedua toolkit GUI memanggil kode Anda HANYA sebagai respons atas aksi pengguna yang nyata -- tetapi cara callback dipasang, dan apa yang diterimanya, berbeda.

	Java: JavaFX	Python: Tkinter
Memasang callback	button.setOnAction(e -> handler()) lambda yang mengimplemen- tasikan EventHandler<ActionEvent>	button.config(command=handler) sebuah callable polos tanpa argumen
Apa yang diterima callback	Sebuah objek ActionEvent (sering tak dipakai, tapi selalu diberikan)	Tidak ada -- callback command= Tkinter tidak menerima argumen sama sekali
Siapa menjalankan event loop	Application.launch() memulai thread rendering + event milik JavaFX sendiri	root.mainloop() memulai event loop Tk sendiri pada thread pemanggil

Tak satu pun toolkit memanggil kode handler Anda sebelum pengguna benar-benar mengklik -- keduanya mengembalikan kendali KE toolkit persis saat handler Anda selesai. "Event handler" bermakna gagasan yang sama di kedua pohon; objek yang diterimanya (atau tidak) dan siapa pemilik loop hanyalah detail toolkit.

Gambar 11.3 — Penanganan event, dibandingkan: setOnAction(...) milik JavaFX dan command=... milik Tkinter sama-sama memasang callback yang dipanggil toolkit hanya sebagai respons atas aksi pengguna yang nyata, tetapi berbeda dalam apa yang diterima callback dan siapa pemilik event loop.

LibraryController.java (kutipan)

```
public class LibraryController {
    private final Library library;
    private final LibraryView view;

    public LibraryController(Library library, LibraryView view) {
        this.library = library;
        this.view = view;
        wireEvents();
        refreshView();
    }

    private void wireEvents() {
        view.getAddButton().setOnAction(e -> handleAddBook());
        view.getBorrowButton().setOnAction(e -> handleBorrowSelected());
        view.getReturnButton().setOnAction(e -> handleReturnSelected());
        view.getRemoveButton().setOnAction(e -> handleRemoveSelected());
    }

    private void handleBorrowSelected() {
        LibraryItem selected = view.getItemList()
            .getSelectionModel().getSelectedItem();
        if (selected == null) {
            view.setStatus("Borrow failed: no item selected.");
            return;
        }
        boolean ok = selected.borrow();
        view.setStatus(ok
            ? "Borrowed: " + selected.getTitle()
            : "Borrow refused: " + selected.getTitle() + " is unavailable "
              + "(already on loan, or a reference-only item).");
        refreshView();
    }

    private void refreshView() {
        view.refresh(library.getAllItems());
    }
}
```

Tata letak yang terlalu sempit bisa diam-diam memotong label kontrolnya sendiri

Versi pertama LibraryView yang dibangun di bab ini memuat ketiga tombol aksi ke dalam satu HBox di dalam panel samping selebar 280 piksel -- kodenya terkompilasi bersih, berjalan tanpa error, dan setiap tombol memicu handler-nya yang benar ketika diklik. Hanya sebuah screenshot dari eksekusi verifikasi tanpa layar (Bagian 11.7) yang mengungkap cacat sungguhannya: "Borrow Selected" tampil sebagai "Borro…", setiap label terpotong karena HBox itu sama sekali tidak memiliki cukup ruang horizontal untuk tiga tombol selebar penuh. Ini bukan bug logika -- compareTo(), borrow(), dan setiap handler di atas sudah benar -- ini cacat TATA LETAK, tak terlihat oleh javac dan tak terlihat lewat pembacaan kode biasa, dan diperbaiki dengan menumpuk tombol-tombol itu dalam VBox alih-alih, dengan borrowButton.setMaxWidth(Double.MAX_VALUE) (dan hal yang sama untuk dua lainnya) sehingga setiap tombol melebar mengisi penuh lebar kontainernya. Bug pada bab GUI tidak selalu ada pada logikanya; sebagian hanya terlihat pada gambar jendela yang sungguh berjalan.

11.6 Titik Masuk Aplikasi: LibraryApp

LibraryApp meng-extend javafx.application.Application, kelas dasar wajib JavaFX untuk titik masuk program GUI apa pun. Method start(Stage stage)-nya berjalan sekali: membangun Model (sebuah Library yang sudah diisi empat item yang sama yang dipakai mata kuliah ini sejak Bab 7-10), membangun View dan Controller yang menghubungkannya ke Model itu, lalu menampilkan Stage (istilah JavaFX untuk jendela tingkat-atas). Setelah stage.show() kembali, kendali berpindah ke event loop milik JavaFX sendiri, dan tidak ada apa pun lagi di start() yang berjalan selama sesi interaktif biasa -- setiap baris perilaku berikutnya datang dari handler Controller yang merespons klik sungguhan.

LibraryApp.java (kutipan)

```
public class LibraryApp extends Application {
    @Override
    public void start(Stage stage) throws IOException {
        Library library = new Library();
        library.addItem(new Book("Clean Code", "Robert C. Martin",
            "9780132350884", 2008));
        library.addItem(new DVD("The Imitation Game",
            "9900000000001", 114));
        library.addItem(new ReferenceBook("Encyclopedia of Computer Science",
            "Ralston & Reilly", "9780123456789", 2003));
        library.addItem(new Book("Algorithms", "Robert Sedgewick",
            "9780321573513", 2011));

        LibraryView view = new LibraryView();
        LibraryController controller = new LibraryController(library, view);

        Scene scene = view.buildScene();
        stage.setTitle("Pustaka -- Library System (Chapter 11)");
        stage.setScene(scene);
        stage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Application.launch() dan event loop milik JavaFX sendiri

launch(args) adalah yang sungguh memulai JavaFX: ia menginisialisasi runtime JavaFX, membuat Stage yang dioper ke start(), lalu memulai thread rendering-dan-event milik toolkit itu sendiri, yang menunggu klik dan mengirimkannya ke handler mana pun yang dipasang dengan setOnAction(...). main() yang memanggil launch(args) lalu tidak melakukan apa-apa lagi adalah normal dan diharapkan -- tidak ada baris kode dalam program ini yang menjalankan sesuatu setara while(true); JavaFX menyediakan loop itu sendiri, dan program hanya pernah bereaksi.

11.7 Menyatukan Semuanya: Memverifikasi GUI Sungguh Berjalan

Driver bab konsol menghasilkan transkrip teks yang trivial untuk direproduksi: jalankan lagi, dapatkan baris System.out yang identik. GUI tidak memiliki output teks setara secara default, sehingga

LibraryApp bab ini tambahan mendukung jalur verifikasi terskrip, digerbangi di belakang system property JVM -Dpustaka.verify=true (dibaca lewat Boolean.getBoolean("pustaka.verify")), yang tidak pernah diambil eksekusi interaktif biasa. Ketika property itu diset, start() memanggil runScriptedVerification(), yang menyimpan screenshot jendela yang baru dibuka, memilih baris list pertama SECARA TERPROGRAM (view.getItemList().getSelectionModel().select(0)), lalu memanggil view.getBorrowButton().fire() -- API milik JavaFX sendiri untuk memicu tombol persis seperti klik mouse sungguhan, berjalan lewat lambda setOnAction(...) yang identik dengan yang dikirim sebuah klik -- lalu menyimpan screenshot kedua sesudahnya.

LibraryApp.java -- runScriptedVerification() (kutipan)

```
private void runScriptedVerification(LibraryView view, Scene scene)
        throws IOException {
    saveSnapshot(scene, "gui_screenshot_1_initial.png");

    view.getItemList().getSelectionModel().select(0);
    view.getBorrowButton().fire();   // jalur kode yang sama persis dengan klik
mouse sungguhan
    System.out.println("status now reads: \"" + view.getStatusLabel().getText()
+ "\"");

    saveSnapshot(scene, "gui_screenshot_2_after_borrow.png");
    Platform.exit();
}

private void saveSnapshot(Scene scene, String fileName) throws IOException {
    WritableImage image = scene.snapshot(null);
    ImageIO.write(SwingFXUtils.fromFXImage(image, null), "png", new
File(fileName));
}
```

Button.fire() tidak mensimulasikan atau mendeskripsikan sebuah klik; ia menjalankan callback setOnAction(...) yang identik dengan yang dipanggil JavaFX ketika mouse sungguh mengklik tombol itu, sehingga screenshot dan teks status yang dihasilkan adalah bukti sungguhan bahwa rantai handler LibraryController -- baca seleksi View, ubah Model lewat selected.borrow(), suruh View menyegarkan tampilan -- bekerja benar dari ujung ke ujung, bukan deskripsi tertulis tentang apa yang diharapkan penulis akan terjadi. Kedua screenshot direproduksi di Lampiran 11.A, berdampingan dengan log teks lengkap yang dihasilkan eksekusi ini.

11.8 Mengkompilasi dan Menjalankan Program Anda

JavaFX dikirim sebagai kumpulan modul terpisah dari JDK itu sendiri (lingkungan mata kuliah ini memakai OpenJFX 11.0.11), sehingga baik mengompilasi maupun menjalankan membutuhkan dua flag tambahan di luar javac/java polos setiap bab sebelumnya: --module-path yang menunjuk ke jar JavaFX, dan --add-modules yang menyebutkan modul spesifik yang dipakai kode bab ini (javafx.controls untuk kontrol UI, javafx.swing untuk SwingFXUtils, hanya dipakai jalur verifikasi kode screenshot-nya).

```
$ javac --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
        LibraryItem.java Renewable.java Book.java DVD.java ReferenceBook.java \
        ItemNotFoundException.java Pair.java LibraryUtils.java Library.java \
        LibraryView.java LibraryController.java LibraryApp.java
$ java --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
       LibraryApp

# eksekusi verifikasi terskrip (menghasilkan kedua screenshot di Lampiran 11.A):
$ java --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
       -Dpustaka.verify=true LibraryApp
```

Memverifikasi GUI sungguhan tanpa layar fisik

Kode bab ini dikompilasi dan dijalankan dalam lingkungan sandbox tanpa layar fisik terpasang, memakai Xvfb, sebuah virtual X11 display server: xvfb-run -a java ... memberi JavaFX sebuah display untuk digambar yang hanya ada di memori, membiarkan program berjalan, mengklik lewat event handler sungguhannya, dan menghasilkan screenshot genuine tanpa monitor. Ini diungkapkan di sini, bukan disamarkan: verifikasinya sepenuhnya nyata -- kode JavaFX yang sama, pengiriman event yang sama, pipeline rendering yang sama yang dipakai eksekusi desktop -- hanya display itu sendiri yang virtual, bukan disimulasikan.

11.9 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Dashboard admin internal Profitina mengikuti pemisahan MVC yang sama yang diajarkan bab ini: sebuah lapisan layanan backend (Model, dalam semangatnya) yang tidak tahu apa-apa tentang bagaimana dashboard mana pun menggambar sebuah chart, sebuah lapisan rendering (View) yang hanya tahu cara menggambar data apa pun yang diberikan kepadanya, dan sebuah lapisan koordinasi tipis (Controller) yang membaca apa yang diklik operator, meminta layanan backend melakukan sesuatu, dan menyuruh lapisan rendering menggambar ulang. Disiplin yang sama yang ditegakkan bab ini pada skala mainan -- Model tidak pernah meng-import apa pun tentang View -- adalah yang membiarkan backend Profitina diuji, bahkan dipakai ulang, sepenuhnya terpisah dari dashboard tertentu mana pun yang dibangun di atasnya.

11.10 Ringkasan Bab dan Poin-Poin Utama

- Pola Model-View-Controller memecah program GUI menjadi Model (data dan aturan, tanpa pengetahuan GUI), View (kontrol yang terlihat, nol logika bisnis), dan Controller (satu-satunya kelas yang boleh bergantung pada keduanya).
- Hierarki Library dan LibraryItem milik Bab 5-10 hanya perlu tepat satu method baru (getAllItems(), aksesor salinan-defensif) agar bisa dipakai dari View GUI -- setiap

pemanggilan borrow()/returnItem()/removeItemOrThrow() dipakai ulang sepenuhnya tanpa perubahan.

- Pemrograman berbasis event membalik alur kendali program konsol: event loop milik JavaFX sendiri, bukan kode programmer, yang memutuskan kapan sebuah handler berjalan, memanggil balik tepat lambda yang dipasang dengan setOnAction(...) untuk kontrol apa pun yang baru saja berinteraksi dengan pengguna.
- Setiap handler Controller mengikuti bentuk tiga-langkah yang sama: baca apa yang sedang ditampilkan View, suruh Model melakukan sesuatu, lalu suruh View menyegarkan -- bentuk yang sama yang dipakai Controller jalur Python bab ini (Bagian 11.5 buku itu) dengan callback command= Tkinter, bukan setOnAction().
- Button.fire() menjalankan jalur kode event-handler yang identik dengan yang dikirim klik mouse sungguhan, membuat eksekusi verifikasi terskrip tanpa layar menjadi bukti sungguhan bahwa GUI-nya bekerja, bukan deskripsi perilaku yang diharapkan -- dan eksekusi verifikasi yang sama itulah yang menangkap satu cacat sungguhan bab ini, sebuah tata letak yang terlalu sempit untuk menampilkan tiga label tombol selebar penuh.

View yang tidak pernah meng-import aturan bisnis Model, dan Model yang tidak pernah meng-import apa pun tentang View, bukanlah cita-cita abstrak di bab ini -- itulah alasan satu-satunya cacat tak-terencana LibraryView adalah masalah lebar piksel, dan tidak pernah satu pun jumlah dolar yang salah, satu pun keputusan peminjaman yang salah, atau satu pun daftar item yang basi.

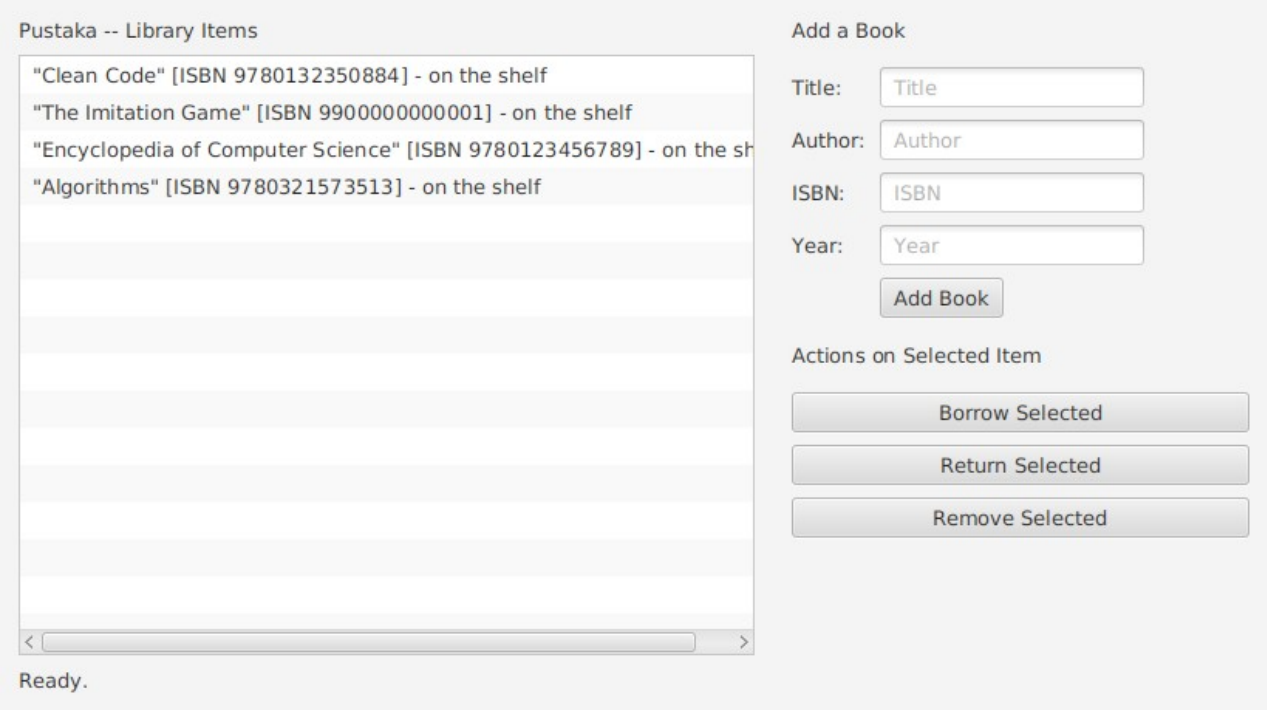
11.11 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

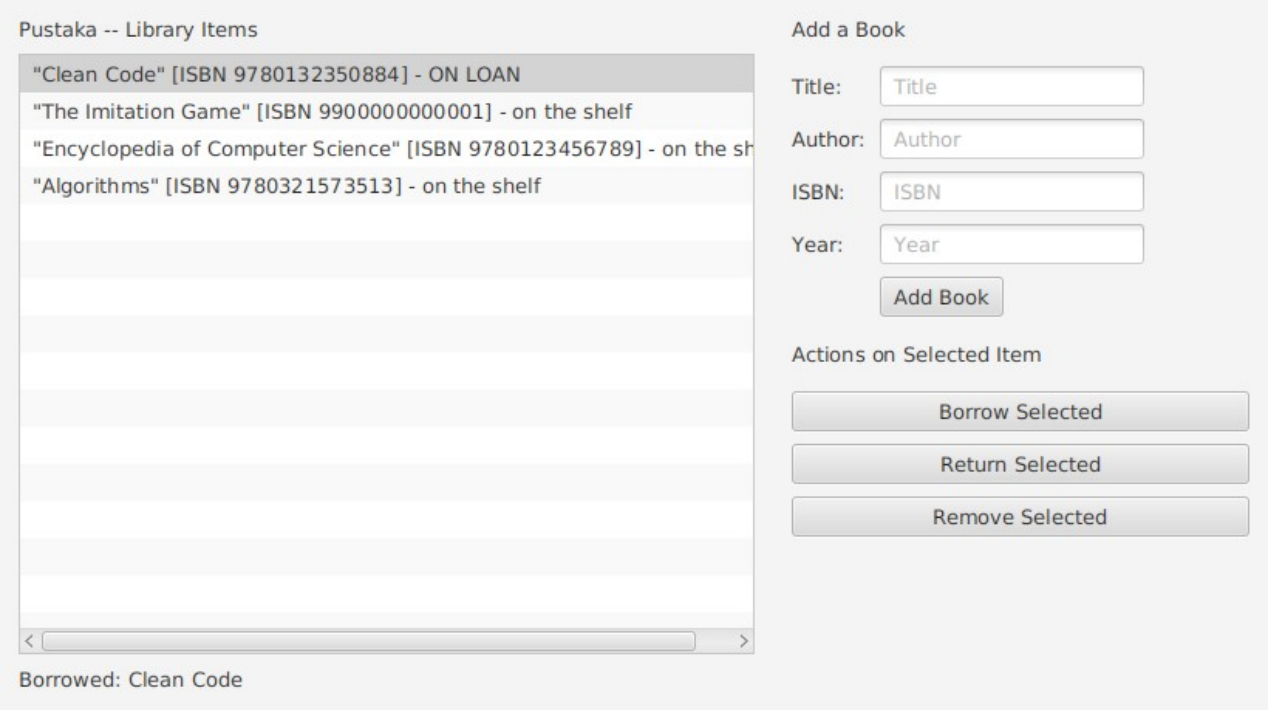
1. LibraryController.handleRemoveSelected() menangkap ItemNotFoundException meskipun ISBN yang dioper ke removeItemOrThrow() datang langsung dari sebuah LibraryItem yang sudah ada di dalam list milik View sendiri. Jelaskan mengapa blok catch itu tetap dibutuhkan agar kodenya terkompilasi, dan mengapa cabang di dalamnya seharusnya tidak pernah sungguh berjalan dalam praktik.
2. Andaikan LibraryView mengekspos library secara langsung (field atau getter publik) alih-alih hanya mengekspos kontrolnya sendiri. Deskripsikan satu perubahan yang bisa dibuat LibraryController yang akan melanggar batas MVC yang ditetapkan bab ini, dan jelaskan apa yang akan salah jika dua View berbeda sama-sama mencoba melakukan ini.
3. view.getBorrowButton().fire() dipakai dalam eksekusi verifikasi bab ini alih-alih memanggil langsung library.findByTitle(...).borrow() dari kode verifikasi. Jelaskan apa yang gagal dibuktikan oleh pemanggilan Model langsung semacam itu yang berhasil dibuktikan fire().
4. Cacat pemotongan-label-tombol di Bagian 11.5 tak terlihat oleh javac dan tak terlihat lewat pembacaan kode biasa. Sebutkan satu jenis cacat GUI lain (bukan terkait tata letak) yang juga akan terkompilasi dan berjalan tanpa error, namun hanya tertangkap dengan melihat screenshot sungguhan.
5. Tuliskan ulang handleBorrowSelected() sehingga, alih-alih memanggil selected.borrow() secara langsung, ia lebih dulu memeriksa selected.isOnLoan() dan menyetel pesan status "Already on loan." sebelum pernah memanggil borrow() sama sekali. Apakah ini akan mengubah antarmuka publik Model? Jelaskan jawaban Anda.

Lampiran 11.A — Log Verifikasi Eksekusi

Berkas lengkap LibraryItem.java, Renewable.java, Book.java, DVD.java, ReferenceBook.java, ItemNotFoundException.java, Pair.java, LibraryUtils.java, Library.java, LibraryView.java, LibraryController.java, dan LibraryApp.java (Code/Java/Chapter11/, disertakan bersama buku ini) dikompilasi dan dijalankan di bawah Xvfb pada OpenJDK 21.0.10 dengan OpenFX 11.0.11 sebelum bab ini ditulis, memakai jalur terskrip -Dpustaka.verify=true yang dideskripsikan Bagian 11.7. Kedua screenshot di bawah adalah output genuine, tak-diedit, dari eksekusi itu -- bukan mockup -- dan log teks di bawahnya direproduksi verbatim (Gambar 11.4 dan 11.5).



Gambar 11.4 — Screenshot 1: jendela segera setelah dibuka. Keempat item menunjukkan "on the shelf"; belum ada yang dipilih.



Gambar 11.5 — Screenshot 2: setelah eksekusi terskrip memilih baris 0 dan memicu Borrow Selected. "Clean Code" tersorot dan kini terbaca "ON LOAN"; label status terbaca "Borrowed: Clean Code".

```
$ javac --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing *.java
(tidak ada error)

$ xvfb-run -a java --module-path /usr/share/openjfx/lib --add-modules
javafx.controls,javafx.swing \
          -Dpustaka.verify=true LibraryApp
--- Chapter 11 GUI verification run ---
Snapshot 1 saved: initial window, 4 items loaded, nothing selected.
Selected list row 0 (programmatically, same selection a click would make).
Fired Borrow Selected button -> status now reads: "Borrowed: Clean Code"
Snapshot 2 saved: after borrowing the selected item.
```

Referensi

[1] Oracle Corporation. "JavaFX Overview." JavaFX Documentation Project. https://openjfx.io/openjfx-docs/ (diakses Agustus 2026).

[2] Oracle Corporation. "Handling JavaFX Events." JavaFX Documentation Project. https://openjfx.io/javadoc/21/javafx.graphics/javafx/event/package-summary.html (diakses Agustus 2026).

[3] G. E. Krasner and S. T. Pope. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System." Journal of Object-Oriented Programming, 1(3), 1988.

[4] Oracle Corporation. "Scene.snapshot(SnapshotParameters, WritableImage)." JavaFX API Documentation. https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/Scene.html (diakses Agustus 2026).

BAB 12 · BAB LATIHAN

Soal Latihan: Kuis 2 — Meninjau Ulang Kuliah 1 Sampai 11

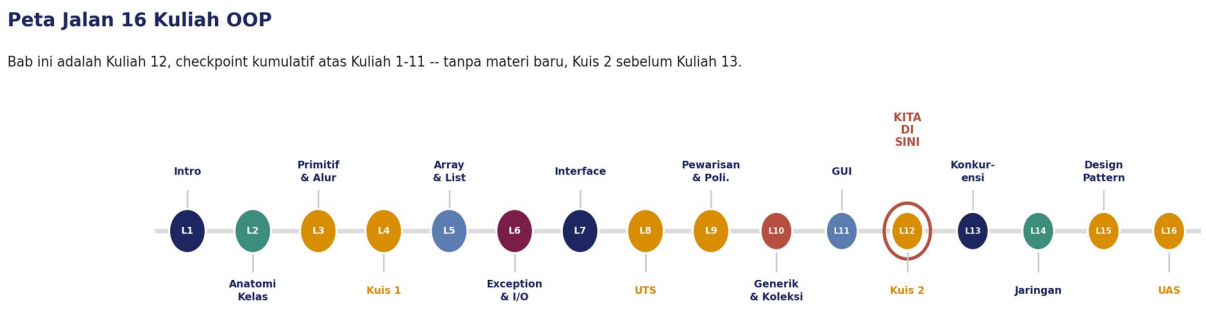
Tidak ada materi baru di sini -- delapan soal yang mencakup semuanya dari enkapsulasi hingga desain Model-View-Controller, meninjau ulang setiap kuliah sejak Kuliah 1 sebelum Kuis 2.

Tujuan Pembelajaran — setelah mengerjakan bab ini Anda akan mampu:

(1) Merancang kelas terenkapsulasi kecil yang menyimpan sepasang nilai dan menurunkan nilai lain darinya sesuai permintaan. (2) Menulis logika alur kontrol yang berperilaku benar pada setiap nilai batas, bukan hanya kasus yang jelas. (3) Membangun string terformat secara efisien dari sebuah list, menggunakan StringBuilder alih-alih konkatenasi berulang. (4) Mendefinisikan dan melempar exception kustom yang membiarkan state sebuah objek tidak tersentuh saat gagal, serta membaca dan menulis berkas log teks polos dengan aman. (5) Merancang kelas abstrak dengan method konkret yang dibangun di atas method abstrak, dan meng-extend kelas dengan pewarisan sungguhan yang memanggil implementasi induk lewat super serta melakukan dispatch polimorfik atas list campuran. (6) Menulis method generik yang dibatasi ke Comparable, membangun indeks menggunakan Map, dan merancang desain Model-View-Controller di mana setiap kelas memiliki satu tanggung jawab yang jelas batasnya.

12.1 Rekap: Kuliah 1–11 Secara Singkat

Kuliah 1 sampai 8 membangun perkakas inti bahasa ini dan sudah ditinjau ulang secara lengkap oleh UTS di Bab 8: pergeseran dari pemikiran prosedural ke berorientasi objek (Kuliah 1), anatomi kelas dan enkapsulasi (Kuliah 2), primitif dan alur kontrol (Kuliah 3), Kuis 1 (Kuliah 4), array, List, dan StringBuilder (Kuliah 5), exception dan I/O berkas (Kuliah 6), interface dan kelas abstrak (Kuliah 7), serta UTS itu sendiri (Kuliah 8). Kuliah 9 membangun langsung di atas kelas abstrak dan interface Kuliah 7 dengan pewarisan sungguhan -- extends, super, method overriding, dan dynamic dispatch yang di-resolve secara polimorfik saat runtime, bukan saat kompilasi. Kuliah 10 menambahkan Generics Java dan Collections Framework, memungkinkan hierarki LibraryItem milik Pustaka disimpan, diurutkan, dan dicari dalam List, Set, atau Map yang type-safe, bukan array mentah. Kuliah 11 membawa Library sepenuhnya keluar dari konsol, membangun LibraryView dan LibraryController di atas event loop milik JavaFX sendiri dengan pola Model-View-Controller -- pola yang sama yang diminta Soal 8 di bawah untuk Anda rancang bagi aplikasi yang jauh lebih kecil (Gambar 12.1).



Gambar 12.1 — Bab ini berada di Kuliah 12 dalam peta jalan 16-kuliah OOP: sebuah checkpoint, bukan topik baru, Kuis 2 sebelum Kuliah 13.

12.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini -- dan bukan apa

Ini adalah Bab Latihan: bab praktik, bukan bab materi baru. Setiap soal di bawah berakar pada materi spesifik dari salah satu dari delapan kuliah sejak Kuliah 1 yang mengajarkan konten baru (lihat Gambar 12.2). Setiap soal disertai PETUNJUK -- dorongan ke arah cara berpikir yang tepat -- tapi sengaja BUKAN solusi lengkap atau kode sumber. Kedelapan soal ini berdiri sendiri, independen dari studi kasus Pustaka yang berjalan di seluruh bab biasa -- persis seperti soal-soal Kuis 1 di Bab 4 dan soal-soal UTS di Bab 8.

Asal Setiap Soal Latihan

Setiap soal di Bagian 12.3 dapat ditelusuri kembali ke salah satu dari delapan kuliah sejak Kuliah 1 yang mengajarkan materi baru.

#	Soal	Berasal dari
1	Kelas Rectangle	K2 -- field, konstruktor, enkapsulasi
2	Kategori BMI	K3 -- alur kendali, kondisi batas
3	Pemformat Baris CSV	K5 -- StringBuilder, array dan list
4	Penjaga Saldo Negatif	K6 -- exception checked kustom, berkas log transaksi
5	Hierarki Shape	K7 -- kelas abstrak, method konkret di atas method abstrak
6	Vehicle & Car	K9 -- pewarisan, polimorfisme, super.describe()
7	maxOf Generik	K10 -- generik terbatas atas Comparable, indeks berbasis Map
8	MVC Counter Mini	K11 -- Model-View-Controller, desain berbasis event

Gambar 12.2 — Setiap soal di Bagian 12.3 dapat ditelusuri kembali ke salah satu dari delapan kuliah sejak Kuliah 1 yang mengajarkan materi baru.

Sebelum Anda mulai

Coba setiap soal sungguh-sungguh dalam proyek .java baru sebelum membaca petunjuknya. Jika buntu, baca hanya petunjuknya -- bukan solusi -- lalu coba lagi. Ini persis disiplin yang akan dihargai oleh Kuis 2 di Bagian 12.4: asesmennya open-book, tapi itu bukan pengganti penalaran Anda sendiri.

12.3 Soal Latihan

12.3.1 Review Field, Konstruktor & Enkapsulasi

Soal 1 ***[Mudah]***. Definisikan kelas Q121Rectangle(double width, double height), hanya menyimpan kedua dimensi, dengan getWidth(), getHeight(), getArea() (width * height), dan getPerimeter() (2 * (width + height)).

Petunjuk

Simpan hanya field width dan height. Baik getArea() maupun getPerimeter() harus menghitung hasilnya sesuai permintaan dari kedua field itu, bukan menyimpan area dan perimeter sebagai field yang bisa jadi tidak sinkron jika width atau height pernah diubah -- persis disiplin yang sama dengan Q81Temperature di Bab 8.

12.3.2 Review Primitif & Alur Kontrol

Soal 2 *[Mudah]*. Tulis kelas Q122BmiCalculator dengan method static String bmiCategory(double weightKg, double heightM) yang menghitung bmi = weightKg / (heightM * heightM) dan mengembalikan "Underweight" untuk bmi < 18,5, "Normal" untuk bmi < 25,0, "Overweight" untuk bmi < 30,0, dan "Obese" jika tidak. Uji pada 18,5, 18,49, 25,0, 24,99, 30,0, dan 29,99.

> **Petunjuk**
>
> Kali ini urutkan perbandingan Anda dari batas terendah ke tertinggi, karena kategorinya dinyatakan dengan < bukan >= -- rantai if/else-if yang kembali begitu satu kondisi cocok. Kasus 18,49/24,99/29,99 sengaja ada untuk menangkap kesalahan < versus <= persis pada nilai batas, yang tidak akan pernah terungkap hanya dengan angka bulat.

12.3.3 Array, List & StringBuilder

Soal 3 *[Sedang]*. Tulis method static String formatCsvRow(String[] fields) yang menggabungkan field dengan koma menjadi satu baris CSV, kecuali field yang mengandung koma harus dibungkus tanda kutip ganda terlebih dahulu (versi sederhana dari aturan quoting CSV yang sesungguhnya). Contoh: {"Ann", "Budi, Jr.", "30"} menjadi Ann,"Budi, Jr.",30. Bangun hasilnya dengan StringBuilder, bukan konkatenasi String berulang.

> **Petunjuk**
>
> Lakukan loop dengan indeks agar Anda tahu apakah sedang berada di field terakhir (tanpa koma di belakangnya). Untuk setiap field, periksa dulu field.contains(",") -- jika benar, bungkus field itu dengan karakter tanda kutip ganda sebelum menambahkannya, jika tidak tambahkan apa adanya.

12.3.4 Penanganan Exception & I/O Berkas

Soal 4 *[Sedang]*. Definisikan kelas exception checked kustom NegativeBalanceException, dan kelas Q124Account(double openingBalance) dengan method void withdraw(double amount) throws NegativeBalanceException, IOException yang melempar NegativeBalanceException (pesannya menyertakan baik jumlah yang diminta maupun saldo saat ini) jika amount melebihi saldo saat ini, dan jika tidak mengurangi amount dari saldo serta menambahkan satu baris "WITHDRAW,amount,balanceAfter" ke berkas log transaksi.

> **Petunjuk**
>
> Periksa kondisi saldo tidak cukup dan lempar exception SEBELUM menyentuh field saldo atau membuka berkas -- penarikan yang gagal harus membiarkan keduanya sepenuhnya tidak tersentuh. Setelah pemeriksaan lolos, perbarui field saldo terlebih dahulu, lalu gunakan try-with-resources dengan BufferedWriter yang dibuka dalam mode append untuk menulis baris log -- persis pola yang dipakai saveToFile di Bab 6 untuk Pustaka dan berkas log Q86 di Bab 8 untuk skor.

12.3.5 Interface & Kelas Abstrak

Soal 5 *[Sedang]*. Definisikan kelas abstrak Q125Shape dengan method abstrak double area(), dan method KONKRET String describe() yang dibangun sepenuhnya dari area() (mis. mengembalikan

sesuatu seperti "This shape covers 28.27 square units."). Lalu definisikan dua subclass konkret: Q125Circle(double radius) dan Q125Square(double side).

> **Petunjuk**
>
> describe() sebaiknya berada di kelas abstrak itu sendiri, bukan di salah satu subclass -- ia sudah sepenuhnya bisa diimplementasikan dalam istilah area() apa pun yang akhirnya disediakan tiap subclass, persis seperti Q87Employee.annualPay() di Bab 8 yang merupakan method konkret dibangun di atas monthlyPay() yang abstrak.

12.3.6 Pewarisan & Polimorfisme

Soal 6 *[Sedang]*. Definisikan kelas Q126Vehicle(String make, String model) dengan method String describe() yang mengembalikan "make model". Lalu definisikan subclass Q126Car(String make, String model, int doorCount) yang meng-OVERRIDE describe() untuk memanggil super.describe() dan menambahkan jumlah pintu, mis. "Toyota Corolla (4 doors)". Terakhir, bangun List<Q126Vehicle> berisi campuran objek Q126Vehicle biasa dan Q126Car, lalu panggil describe() pada masing-masing dalam sebuah loop.

> **Petunjuk**
>
> Q126Car.describe() harus memanggil super.describe() alih-alih memformat ulang make dan model sendiri -- itulah pewarisan yang menggunakan ulang logika induk, bukan menduplikasinya. Loop atas List<Q126Vehicle> adalah dispatch polimorfik: meskipun tipe referensinya Q126Vehicle, setiap pemanggilan describe() di-resolve saat runtime ke kelas mana pun objek sesungguhnya berasal.

12.3.7 Generics & Collections Framework

Soal 7 *[Sulit]*. Tulis method static generik <T extends Comparable<T>> T maxOf(List<T> items) yang mengembalikan elemen terbesar dari items (melempar IllegalArgumentException untuk list kosong). Lalu tulis method static Map<Character, List<String>> groupByFirstLetter(List<String> words) yang mengelompokkan kata ke dalam Map berdasarkan huruf pertamanya (dikapitalkan).

> **Petunjuk**
>
> Batasan <T extends Comparable<T>> inilah yang memungkinkan maxOf memanggil item.compareTo(other) tanpa cast yang unchecked -- tanpa batasan itu, T bisa berupa tipe apa pun, termasuk yang sama sekali tidak memiliki urutan. Untuk groupByFirstLetter, gunakan Map.computeIfAbsent(key, k -> new ArrayList<>()).add(word) agar Anda tidak perlu pemeriksaan containsKey terpisah sebelum menambahkan ke list sebuah grup.

12.3.8 GUI & MVC

Soal 8 *[Sulit]*. Rancang aplikasi counter MVC mini: kelas Model Q128CounterModel dengan increment(), decrement(), dan getValue(); kelas View yang menampilkan nilai saat ini dalam sebuah label serta menyediakan tombol "+" dan tombol "-"; dan kelas Controller yang menjadi SATU-SATUNYA kelas yang memegang referensi ke Model maupun View, menghubungkan setiap tombol ke method Model yang sesuai lalu me-refresh label View. Sketsakan field dan signature method ketiga kelas itu -- Anda tidak perlu GUI yang sungguh berjalan untuk menjawab soal ini.

Petunjuk

Persis seperti LibraryView/LibraryController di Bab 11: Model harus nol import terhadap toolkit GUI Anda, View harus nol logika bisnis (tidak ada increment yang terjadi di dalam View), dan hanya Controller yang boleh meng-import dan bergantung pada keduanya. Jika penangan klik tombol mana pun mengubah count internal Q128CounterModel secara langsung alih-alih memanggil increment()/decrement(), maka View itu telah merusak enkapsulasi persis seperti yang diperingatkan Bab 11.

12.4 Format Kuis 2: Open-Book, Individu, Timed

Kuis 2 adalah penilaian take-home open-book, individu, dan timed (kurang lebih 120 menit) yang mencakup persis delapan jenis soal yang direview di bab ini -- sebuah checkpoint kumulatif atas semuanya sejak Kuliah 1, bukan hanya materi sejak UTS. Penilaian dilakukan per-soal, bukan semua-atau-tidak-sama-sekali: nilai parsial diberikan untuk kelas yang kompilasi dan menangani kasus umum dengan benar sekalipun satu edge case terlewat.

Open-book berarti referensi, bukan kolaborator

Anda boleh berkonsultasi dengan buku teks, catatan Anda sendiri, dan slide kuliah saat menjawab. Anda TIDAK BOLEH berkolaborasi dengan teman sekelas atau mengumpulkan kode yang bukan benar-benar milik Anda sendiri -- kuis open-book yang timed menguji apakah Anda bisa MENERAPKAN apa yang sudah Anda pelajari, tanpa pengawasan, dalam batas waktu.

Kriteria	Yang dinilai	Bobot
Kebenaran output	Apakah kelas, method, atau fungsi menghasilkan hasil yang persis sesuai harapan untuk setiap kasus uji yang diberikan, termasuk nilai batas?	45%
Desain kelas, interface & generik	Apakah field bersifat private, state hanya bisa dijangkau lewat method, dan apakah pemisahan kelas abstrak (Soal 5) serta batasan generik (Soal 7) digunakan dengan benar?	25%
Disiplin exception, I/O & MVC	Apakah exception Soal 4 membawa pesan yang berguna dan membiarkan state akun tidak tersentuh saat gagal, apakah I/O berkas-nya menggunakan try-with-resources dengan benar, dan apakah sketsa Soal 8 menjaga Model, View, dan Controller terpisah dengan bersih?	15%
Kejelasan kode & kompilasi bersih	Apakah kode mudah dibaca, penamaannya masuk akal, dan bebas dari kode mati -- serta berkompilasi dengan javac tanpa error?	15%

12.5 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Kedelapan soal bab ini, digabungkan, mendekati versi mini dari apa yang sungguh dibutuhkan satu fitur di Profitina: satu atau dua kelas terenkapsulasi kecil (Soal 1), logika yang diuji-batas dengan cermat (Soal 2), penanganan teks yang efisien untuk sebuah laporan atau respons API (Soal 3), operasi yang harus gagal dengan aman tanpa merusak state yang sudah ada, dicatat secara persisten ke disk (Soal 4), sebuah kontrak bersama yang diimplementasikan oleh beberapa bentuk data konkret (Soal 5), hierarki yang menggunakan ulang perilaku induk alih-alih menduplikasinya (Soal 6), utilitas generik yang bisa dipakai lintas banyak tipe data yang tidak berkaitan ditambah indeks berbasis Map untuk

pencarian cepat (Soal 7), dan pemisahan Model-View-Controller yang membuat logika render sebuah dashboard tidak peduli terhadap layanan backend yang memasoknya (Soal 8, pemisahan yang sama yang digunakan dashboard admin Profitina sendiri). Kuis yang hanya menguji satu dari kemampuan ini akan melewatkan betapa seringnya fitur nyata membutuhkan beberapa di antaranya bekerja bersama dalam satu potongan kode kecil yang sama.

12.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru -- ini adalah checkpoint kumulatif yang mencakup Kuliah 1 sampai 11.
- Delapan soal berdiri sendiri mencakup seluruh rentang yang direview: enkapsulasi (K2), batas alur kontrol (K3), StringBuilder (K5), exception kustom dan I/O berkas (K6), kelas abstrak (K7), pewarisan dan polimorfisme (K9), generics dan Collections Framework (K10), serta desain Model-View-Controller (K11).
- Setiap soal menyertakan petunjuk, bukan solusi -- mengerjakan penalaran dan kodenya sendiri adalah intinya.
- Kuis 2 adalah penilaian take-home open-book, individu, dan timed: referensi diperbolehkan, tapi kodenya harus milik Anda sendiri, dan kebenaran pada setiap kasus uji yang diberikan (termasuk nilai batas) adalah porsi terbesar nilai.

Kelas yang berkompilasi tidak sama dengan kelas yang benar -- dan desain yang terlihat rapi di atas kertas tidak sama dengan desain yang sungguh menjaga Model, View, dan Controller-nya agar tidak saling bergantung.

Lampiran 12.A — Checklist Self-Check (Tidak Dinilai)

Sebelum mengumpulkan jawaban apa pun -- pada soal bab ini atau pada Kuis 2 itu sendiri -- jalankan lewat checklist ini. Butuh waktu beberapa menit dan menangkap sebagian besar kesalahan yang paling sering dilihat penilai.

- Apakah setiap kelas dan method berkompilasi dengan javac tanpa error dan tanpa warning?
- Apakah setiap field bersifat private, dengan setiap nilai yang dibutuhkan pemanggil diekspos hanya lewat method?
- Sudahkah Anda menguji nilai batas persis Soal 2 (18,49, 24,99, 29,99), bukan hanya angka bulat yang bisa menyembunyikan bug < versus <=?
- Apakah withdraw() pada Soal 4 membiarkan field saldo dan berkas log sepenuhnya tidak tersentuh ketika melempar NegativeBalanceException?
- Apakah method abstrak dan method konkret Soal 5 berada pada kelas abstrak yang SAMA, dengan method konkret memanggil yang abstrak alih-alih menduplikasi logika di setiap subclass?
- Apakah Q126Car.describe() pada Soal 6 memanggil super.describe() alih-alih memformat ulang make dan model sendiri?
- Apakah maxOf() pada Soal 7 hanya berkompilasi untuk tipe yang meng-implement Comparable, dan apakah groupByFirstLetter() menghindari pemeriksaan containsKey terpisah sebelum menambahkan ke sebuah grup?
- Sudahkah Anda membaca ulang kode Anda sendiri seolah-olah Anda penilai skeptis yang mencari satu input yang bisa merusaknya?

Referensi

[1] Format penilaian bab latihan ini (open-book, individu, timed) dimodelkan dari Kuis 2 nyata mata kuliah ini (EF234302 Pemrograman Berorientasi Objek), sebuah proyek kelompok membangun sistem prediksi teks T9. Tiga bagian pertama proyek itu berpusat pada tree rekursif bercabang banyak kustom, binary search atas array terurut, dan pencarian Map multi-nilai -- kelas materi Struktur Data yang sama yang sudah ditempatkan UTS nyata pembaruan ini (lihat Referensi [1] Bab 8) di luar cakupan OOP sendiri, sehingga bagian-bagian itu pun tidak digunakan ulang di sini. Bagian keempatnya, meskipun demikian, membangun GUI keypad ponsel menggunakan pola Model-View-Controller -- pola yang sama yang sungguh diajarkan pembaruan ini di Bab 11 -- sehingga Soal 8 di atas mengambil tema GUI/MVC-nya langsung dari bagian itu, sementara ketujuh soal lainnya dirancang baru untuk menguji tujuh area kuliah lain yang perlu dicakup Kuis 2 ini.

[2] Oracle Corporation. "Interfaces and Inheritance." The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/IandI/ (diakses Agustus 2026).

[3] Oracle Corporation. "Lesson: Generics." The Java Tutorials. https://docs.oracle.com/javase/tutorial/extra/generics/ (diakses Agustus 2026).

BAB 13

Konkurensi: Thread, Race Condition, dan Sinkronisasi

Setiap bab sejauh ini menjalankan satu bagian kode pada satu waktu, dalam satu urutan, yang sepenuhnya ditentukan oleh program itu sendiri. Sistem nyata -- termasuk kios publik Pustaka sendiri -- tidak memiliki kemewahan itu: beberapa patron bisa bertindak pada perpustakaan di saat yang persis sama. Bab ini menunjukkan, dengan bug nyata yang bisa Anda reproduksi sendiri, mengapa itu mengubah segalanya tentang bagaimana state bersama harus ditulis -- termasuk satu kejutan jujur tentang betapa sulitnya bug khusus ini untuk diamati secara alami di Python (Gambar 13.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan mengapa operasi sesederhana count++ tidak atomik, dan mendeskripsikan secara konkret bagaimana dua thread bisa berselang-seling sehingga sebuah update hilang. (2) Membuat dan memulai banyak Thread untuk menjalankan pekerjaan secara konkuren, dan meng-join()-nya untuk menunggu semuanya selesai. (3) Menggunakan keyword synchronized untuk menegakkan mutual exclusion pada state bersama, dan menjelaskan persis apa yang dihalangi ketika sebuah thread lain 'memperoleh lock'. (4) Menggunakan thread pool ExecutorService untuk menjalankan banyak tugas singkat tanpa mengelola satu objek Thread secara manual per tugas. (5) Membaca hasil eksekusi program nyata dan membedakan race condition yang sekadar mungkin secara teoretis dari yang benar-benar teramati, dengan angka pasti yang menyertainya.

13.1 Rekap: Pustaka Sejauh Ini

Bab 9 menggeneralisasi Library agar bisa menyimpan LibraryItem mana pun secara polimorfik. Bab 10 menambahkan Generics dan Collections Framework. Bab 11 memberi Library antarmuka berjendela nyata yang dibangun di atas pola Model-View-Controller. Bab 12 adalah checkpoint, bukan materi baru. Setiap bab itu, dan setiap bab sebelumnya, berjalan pada satu thread: satu call stack, satu urutan eksekusi, sepenuhnya bisa diprediksi dari membaca kode sumber dari atas ke bawah. Bab ini adalah pertama kalinya hal itu berhenti berlaku.

Gambar 13.1 — Peta jalan 16-kuliah OOP. Bab ini adalah Kuliah 13: menjalankan lebih dari satu bagian logika Library pada saat yang sama, dengan aman.

13.2 Mengapa Konkurensi: Thread dan State Bersama

Sebuah Thread adalah jalur eksekusi independen di dalam program yang sama yang sedang berjalan, berbagi objek dan memori yang sama dengan setiap thread lain yang dibuat program itu. Rencana deployment nyata Pustaka (terlepas dari GUI Bab 11) membayangkan beberapa kios publik, masing-masing membiarkan patron berbeda meminjam item pada saat yang sama -- dan jika kios-kios itu masing-masing didukung oleh thread-nya sendiri, semuanya bisa berakhir memanggil method pada objek bersama yang persis sama pada instan yang persis sama. Itulah situasi yang dirancang untuk diungkap oleh kelas baru bab ini, BorrowCounter: satu hitungan bersama berapa banyak item yang telah dipinjam di seluruh perpustakaan hari ini, diperbarui dari banyak thread sekaligus.

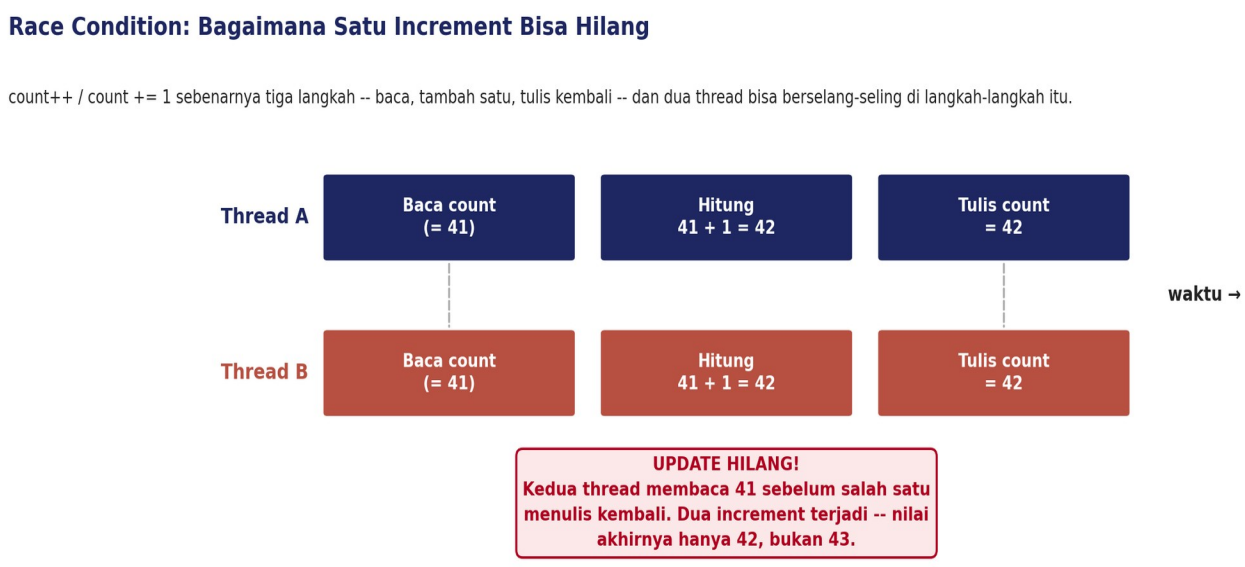
BorrowCounter.java -- kontrak minimal untuk state bersama

```
public interface BorrowCounter {
    void increment();
    int getCount();
}
```

Dua kelas mengimplementasikan interface ini. Salah satunya punya bug yang tak terlihat hanya dengan membaca kode sumber -- ia hanya muncul ketika thread sungguhan benar-benar menjalankannya.

13.3 Race Condition: Bug Nyata yang Bisa Direproduksi

count++ (dan count += 1) terlihat seperti satu operasi tunggal dan tak terbagi dalam kode sumber Java. Ternyata bukan. JVM melakukannya sebagai tiga langkah terpisah: membaca nilai count saat ini, menambahkan satu, dan menulis nilai baru kembali. Tidak ada yang mencegah dua thread berselang-seling di tiga langkah itu -- dan ketika itu terjadi, satu increment hilang secara diam-diam (Gambar 13.2).



Gambar 13.2 — Dua thread sama-sama membaca count sebagai 41 sebelum salah satu menulis kembali; keduanya menghitung 42 dan menulis 42. Salah satu dari dua increment tidak pernah terjadi, dan tidak ada apa pun dalam program yang memunculkan error untuk mengatakannya.

UnsafeBorrowCounter.java -- implementasi naif yang buggy

```
public class UnsafeBorrowCounter implements BorrowCounter {
    private int count = 0;

    @Override
    public void increment() {
        count++;
    }

    @Override
    public int getCount() {
        return count;
    }
}
```

count++ berkompilasi bersih, berjalan tanpa error, dan tetap salah

Tidak ada exception, tidak ada warning, tidak ada crash -- UnsafeBorrowCounter berperilaku sempurna di bawah satu thread, dan javac sama sekali tidak punya komentar apa pun soal itu. Bug-nya hanya ada di celah antara membaca dan menulis, dan celah itu hanya pernah terungkap dengan menjalankan thread konkuren sungguhan terhadapnya, persis yang dilakukan program driver di Bagian 13.7.

13.4 Sinkronisasi: Mutual Exclusion dengan synchronized

Menandai sebuah method synchronized memberitahu JVM untuk memperoleh lock intrinsik (monitor) milik objek pemanggil sebelum menjalankan isi method, dan melepaskannya hanya ketika method itu kembali. Selagi satu thread memegang lock itu, setiap thread lain yang memanggil method synchronized pada objek yang SAMA akan diblokir -- ia hanya menunggu -- sampai lock itu dilepaskan. Itu mengubah 'baca, tambah satu, tulis kembali' menjadi operasi yang berperilaku atomik dari sudut pandang thread lain mana pun, meskipun di baliknya tetap tiga instruksi JVM.

SafeBorrowCounter.java -- implementasi yang diperbaiki

```
public class SafeBorrowCounter implements BorrowCounter {
    private int count = 0;

    @Override
    public synchronized void increment() {
        count++;
    }

    @Override
    public synchronized int getCount() {
        return count;
    }
}
```

getCount() juga butuh synchronized, bukan cuma increment()

Men-synchronized-kan hanya increment() akan menghentikan dua increment berselang-seling SATU SAMA LAIN, tapi thread yang memanggil getCount() selagi thread lain sedang di tengah increment tetap bisa membaca nilai yang sedang berubah -- bahaya yang lebih halus namun tetap nyata, disebut masalah visibility. Men-synchronized-kan kedua method pada lock objek yang sama menutup kedua celah sekaligus.

13.5 Thread Pool: ExecutorService

Membuat dan memulai satu objek Thread per tugas, seperti yang dilakukan dua demo pertama Bagian 13.7, tidak dapat diskalakan melampaui segelintir tugas -- setiap Thread membawa overhead memori dan penjadwalan OS yang nyata. ExecutorService, dari java.util.concurrent, sebaliknya mengelola pool tetap dan bisa dipakai ulang dari thread pekerja: submit() menyerahkan sebuah tugas, dan thread pool mana pun yang paling dulu bebas akan mengambilnya, tanpa objek Thread baru dibuat per tugas (Gambar 13.3).

Mutual Exclusion, Dibandingkan: Dua Cara Menyatakan Hal yang Sama

Kedua bahasa memungkinkan Anda menandai "hanya satu thread boleh menjalankan ini pada satu waktu" -- mekanisme dan sintaksnya berbeda.

	Java	Python
Mendeklarasikan mutual exclusion	synchronized void increment() { ... } sebuah keyword pada method itu sendiri	with self._lock: self._count += 1 objek threading.Lock yang eksplisit
Apa yang dipegang selagi di dalam	Lock intrinsik (monitor) milik objek -- bawaan setiap objek	Objek Lock apa pun yang Anda buat dan pilih untuk diperoleh
Thread pool untuk banyak tugas	ExecutorService + Executors.newFixed-ThreadPool(n)	concurrent.futures.ThreadPoolExecutor(max_workers=n)

Tak satu pun mekanisme menghilangkan bahaya yang mendasarinya secara ajaib -- keduanya bekerja dengan membuat setiap thread LAIN menunggu gilirannya pada satu baris kode yang menyentuh nilai bersama, sehingga "baca, tambah satu, tulis kembali" menjadi efektif satu langkah dari sudut pandang thread lain mana pun.

Gambar 13.3 — Mutual exclusion dan thread pool, dibandingkan lintas kedua jalur yang dipakai mata kuliah ini.

Menyerahkan pekerjaan ke thread pool tetap

```
ExecutorService pool = Executors.newFixedThreadPool(4);
for (int i = 0; i < KIOSK_COUNT; i++) {
    pool.submit(new BorrowWorker(counter, ITERATIONS));
}
pool.shutdown();
pool.awaitTermination(1, TimeUnit.MINUTES);
```

> **shutdown() tidak menghentikan pekerjaan yang sudah diserahkan**
>
> pool.shutdown() memberitahu pool untuk tidak menerima tugas BARU dan untuk mengakhiri thread pekerjanya begitu setiap tugas yang sudah diserahkan selesai -- ini bukan penghentian mendadak. awaitTermination() kemudian memblokir thread pemanggil sampai penyelesaian itu selesai (atau timeout yang diberikan habis), yang membuat program driver bisa membaca hitungan akhir dengan aman segera sesudahnya.

13.6 Konkurensi dan Library: BorrowCounter sebagai Studi Kasus

BorrowCounter sengaja dibuat sebagai kelas kecil yang berdiri sendiri, bukan perubahan pada Library itu sendiri -- persis keputusan yang sama yang diambil Bab 11 ketika menambahkan getAllItems() alih-alih menulis ulang internal Library untuk sebuah GUI. Pelajaran yang diajarkan bab ini (state mutable bersama butuh sinkronisasi eksplisit begitu lebih dari satu thread bisa menjangkaunya) berlaku pada field Library sendiri persis sebanyak ia berlaku pada satu int milik BorrowCounter; BorrowCounter hanya mengisolasi pelajaran itu dari setiap perhatian lain yang sudah dicakup mata kuliah ini, sehingga Bagian 13.7 bisa mendemonstrasikannya secara terisolasi.

BorrowWorker.java -- mensimulasikan satu kios patron

```
public class BorrowWorker implements Runnable {
    private final BorrowCounter counter;
    private final int iterations;

    public BorrowWorker(BorrowCounter counter, int iterations) {
        this.counter = counter;
        this.iterations = iterations;
    }

    @Override
    public void run() {
        for (int i = 0; i < iterations; i++) {
            counter.increment();
        }
    }
}
```

13.7 Program Driver: Tiga Demo, Berturut-turut

ConcurrencyDemo.java mensimulasikan 8 kios patron, masing-masing memanggil increment() 200.000 kali -- sehingga hitungan akhir yang secara matematis benar selalu 8 * 200.000 = 1.600.000, apa pun BorrowCounter yang dipakai. Demo 1 menjalankan beban kerja itu terhadap UnsafeBorrowCounter dengan 8 Thread yang dibuat manual; Demo 2 menjalankan beban kerja yang identik terhadap SafeBorrowCounter dengan cara yang sama; Demo 3 menjalankannya sekali lagi terhadap SafeBorrowCounter, tapi lewat pool ExecutorService 4-thread alih-alih 8 Thread yang dikelola manual.

ConcurrencyDemo.java -- menjalankan demo unsafe dengan thread manual (diringkas)

```
private static int runWithManualThreads(BorrowCounter counter) throws
InterruptedException {
    Thread[] kiosks = new Thread[KIOSK_COUNT];
    for (int i = 0; i < KIOSK_COUNT; i++) {
        kiosks[i] = new Thread(new BorrowWorker(counter, ITERATIONS));
    }
    for (Thread kiosk : kiosks) { kiosk.start(); }
    for (Thread kiosk : kiosks) { kiosk.join(); }
    return counter.getCount();
}
```

start() memulai thread; memanggil run() secara langsung tidak

kiosk.start() meminta JVM untuk memulai thread eksekusi yang sungguh baru, yang kemudian memanggil run() dengan sendirinya. Memanggil kiosk.run() secara langsung hanya akan mengeksekusi loop pada thread SAAT INI, berurutan, tanpa konkurensi sama sekali -- kesalahan umum yang mudah dilakukan dan menghasilkan kode yang berkompilasi bahkan menghasilkan angka yang benar, karena alasan yang salah.

13.8 Mengompilasi dan Menjalankan Program Anda

Kompilasi javac setiap berkas .java di folder Bab 13, lalu java ConcurrencyDemo.

```
$ javac BorrowCounter.java UnsafeBorrowCounter.java SafeBorrowCounter.java \
      BorrowWorker.java ConcurrencyDemo.java
$ java ConcurrencyDemo
```

13.9 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Backend Profitina melayani banyak permintaan konkuren sekaligus, dan beberapa counter miliknya sendiri -- rate limit, tally permintaan yang sedang berjalan, statistik cache hit/miss -- menghadapi persis bahaya bab ini: sebuah counter bersama yang naif diperbarui dari banyak thread penangan permintaan tanpa sinkronisasi akan diam-diam under-count di bawah beban produksi nyata, persis seperti UnsafeBorrowCounter under-count di sini. Perbaikan pada kode produksi adalah gagasan yang sama yang diajarkan bab ini pada skala yang jauh lebih kecil: baik mutual exclusion eksplisit di sekitar state bersama, atau (lebih sering, pada skala Profitina) sebuah struktur data aman-konkurensi khusus dari pustaka standar platform yang sudah menyelesaikan masalah persis ini sekali, dengan benar, sehingga fitur individual tidak perlu menyelesaikannya lagi.

13.10 Ringkasan Bab dan Poin-Poin Utama

- count++ / count += 1 sebenarnya tiga langkah -- baca, tambah satu, tulis kembali -- dan TIDAK atomik; dua thread yang berselang-seling di langkah-langkah itu bisa diam-diam kehilangan sebuah update.
- Pengujian bab ini sendiri mereproduksi kehilangan itu persis pada eksekusi nyata: dari 1.600.000 increment yang diharapkan, hitungan akhir sesungguhnya UnsafeBorrowCounter kurang dari itu -- lihat Lampiran 13.A untuk angka-angka verbatim-nya.
- synchronized pada sebuah method memperoleh lock intrinsik milik objek pemanggil selama durasi method itu; setiap thread lain yang memanggil method synchronized pada objek yang SAMA diblokir sampai lock itu dilepaskan.
- Thread pool ExecutorService menjalankan banyak tugas singkat lewat sekumpulan kecil thread pekerja tetap yang bisa dipakai ulang, alih-alih satu objek Thread per tugas.
- BorrowCounter mengisolasi pelajaran bab ini dari segala sesuatu yang lain yang sudah dilakukan Pustaka -- tapi bahaya yang sama berlaku pada state mutable bersama apa pun yang bisa dijangkau lebih dari satu thread, termasuk field Library sendiri.

Bug yang tidak pernah muncul di bawah satu thread, dan yang tidak dikomentari apa pun oleh javac, bukanlah bug yang tidak ada -- ia adalah bug yang sekadar belum pernah ditanyai satu pertanyaan yang mengungkapkannya.

13.11 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. SafeBorrowCounter men-synchronized-kan baik increment() maupun getCount(). Jelaskan, dengan kata-kata Anda sendiri, apa yang bisa salah jika hanya increment() yang di-synchronized dan getCount() dibiarkan sebagai method biasa yang tidak di-synchronized.
2. Tulis ulang BorrowWorker sehingga, alih-alih memanggil counter.increment() dalam loop, setiap worker membangun int total lokalnya sendiri dan hanya memanggil counter.increment() sekali di paling akhir (dalam loop dari 0 sampai total). Apakah versi ini masih bisa kehilangan update terhadap UnsafeBorrowCounter? Mengapa atau mengapa tidak?
3. Demo 3 ConcurrencyDemo menggunakan Executors.newFixedThreadPool(4) untuk 8 kios. Jelaskan apa yang terjadi pada tugas BorrowWorker ke-5, ke-6, ke-7, dan ke-8 selagi keempat thread pool sibuk dengan 4 yang pertama.
4. Misalkan dua objek BorrowCounter yang berbeda (bukan objek yang sama) masing-masing diakses oleh thread khususnya sendiri, tanpa ada thread yang pernah menyentuh objek yang lain. Apakah sinkronisasi dibutuhkan di sini? Jelaskan mengapa atau mengapa tidak, dengan merujuk kembali pada apa yang sesungguhnya dikunci oleh synchronized.
5. Catatan kejujuran bab ini di unsafe_borrow_counter.py (jalur Python) menjelaskan bahwa race condition tidak muncul dengan andal tanpa perubahan artifisial. Jelaskan, dengan kata-kata Anda sendiri, mengapa bug yang LEBIH umum (pada kode Java produksi nyata yang berjalan) tetap bisa dianggap 'nyata' meskipun tidak muncul pada setiap eksekusi tunggal.

Lampiran 13.A — Log Verifikasi Eksekusi

Berkas BorrowCounter.java, UnsafeBorrowCounter.java, SafeBorrowCounter.java, BorrowWorker.java, dan ConcurrencyDemo.java yang lengkap (Code/Java/Chapter13/, disediakan bersama buku ini) dikompilasi dan dieksekusi pada OpenJDK sebelum bab ini ditulis, untuk memastikan kode itu benar, bukan sekadar 'terlihat benar'. Output direproduksi verbatim di bawah ini dari satu eksekusi nyata.

Angka update-hilang yang persis akan berbeda jika Anda menjalankannya sendiri

Race condition, menurut definisinya, adalah soal timing -- menjalankan ulang ConcurrencyDemo sangat mungkin menunjukkan angka update hilang yang BERBEDA dari eksekusi yang ditangkap di bawah ini, bahkan mungkin nol pada eksekusi yang kurang beruntung (atau beruntung). Yang TIDAK diharapkan berubah adalah hasil kualitatifnya: Demo 1 (unsafe) secara andal jatuh di bawah 1.600.000 di seluruh eksekusi berulang selama pengujian buku ini sendiri, sementara Demo 2 dan Demo 3 (keduanya safe) cocok dengan 1.600.000 persis, setiap saat.

```
$ javac BorrowCounter.java UnsafeBorrowCounter.java SafeBorrowCounter.java \
      BorrowWorker.java ConcurrencyDemo.java
(tidak ada error)

$ java ConcurrencyDemo
Simulating 8 patron kiosks, 200000 borrow operations each.
Mathematically correct final count: 1600000

--- Demo 1: UnsafeBorrowCounter (no synchronization) ---
Actual final count:                1499514
Lost updates:                      100486  <-- a real race condition, not a
typo

--- Demo 2: SafeBorrowCounter (synchronized increment()) ---
Actual final count:                1600000
Matches expected:                  true

--- Demo 3: SafeBorrowCounter via a fixed thread pool (ExecutorService) ---
Actual final count:                1600000
Matches expected:                  true
```

Referensi

[1] Oracle Corporation. "Defining and Starting a Thread" dan "Synchronization." The Java Tutorials, Concurrency lesson. https://docs.oracle.com/javase/tutorial/essential/concurrency/ (diakses Agustus 2026).

[2] Oracle Corporation. "Executor Interfaces." The Java Tutorials, Concurrency lesson. https://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html (diakses Agustus 2026).

[3] B. Goetz et al. Java Concurrency in Practice. Addison-Wesley, 2006, Bab 2 ("Thread Safety").

BAB 14

Networking: Socket, Klien, dan Server

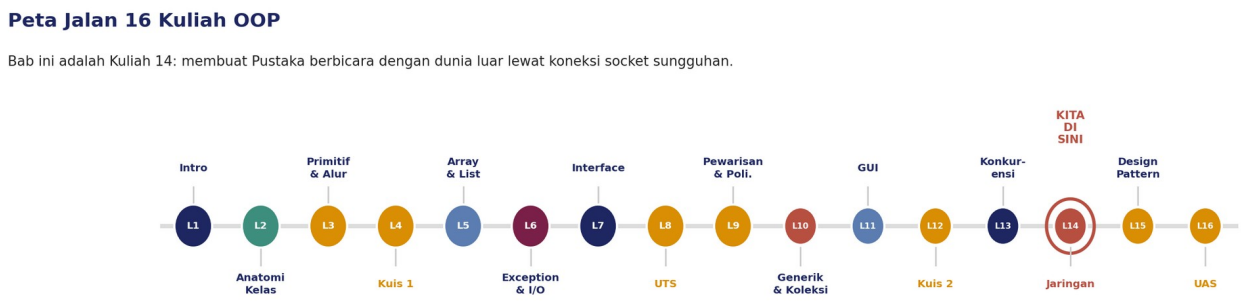
Setiap bab sejauh ini berjalan sepenuhnya pada satu mesin, hanya berbicara dengan dirinya sendiri. Bab ini membuat Pustaka berbicara dengan dunia luar: server TCP nyata yang mendengarkan pada sebuah port, klien TCP nyata yang terhubung kepadanya lewat jaringan, dan protokol request/response nyata yang berjalan di antara keduanya -- dibangun, secara sengaja, di atas Thread yang persis sama yang diperkenalkan mata kuliah ini untuk alasan yang sama sekali berbeda di Bab 13 (Gambar 14.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan model client-server: sebuah server yang mendengarkan pada port tetap, dan klien mana pun yang masing-masing membuka koneksinya sendiri kepadanya. (2) Membuka ServerSocket yang mendengarkan, meng-accept() koneksi masuk, dan membaca dari serta menulis ke stream milik sebuah Socket. (3) Merancang protokol request/response berbasis baris yang minimal, dan mengimplementasikan baik sisi server maupun sisi klien-nya. (4) Menjelaskan mengapa menyerahkan setiap koneksi yang diterima ke Thread-nya sendiri memungkinkan sebuah server melayani banyak klien sekaligus, dan menghubungkan keputusan itu secara eksplisit kembali ke pelajaran konkurensi Bab 13. (5) Membaca log eksekusi client-server nyata dan membedakan apa yang dijamin protokol dari apa yang kebetulan dihasilkan satu eksekusi tertentu.

14.1 Rekap: Pustaka Sejauh Ini

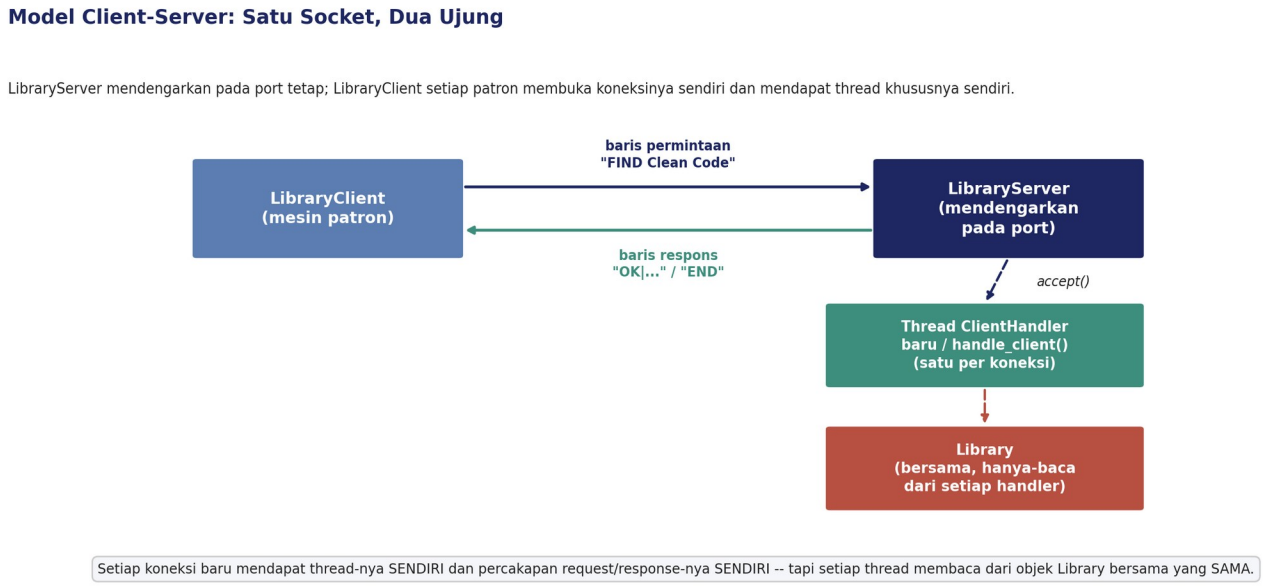
Bab 11 memberi Library antarmuka berjendela nyata. Bab 13 membuat kode terkait-Library berjalan pada lebih dari satu thread sekaligus, menggunakan studi kasus BorrowCounter yang kecil dan berdiri sendiri untuk mengungkap race condition nyata dan memperbaikinya dengan dua cara berbeda. Setiap bab itu, meski begitu, tetap berjalan pada satu mesin: proses mana pun yang menjalankan kode Pustaka adalah satu-satunya proses yang terlibat. Bab ini adalah pertama kalinya hal itu berhenti berlaku -- sebuah proses LibraryServer dan sebuah proses LibraryClient, berjalan sebagai dua program terpisah (berpotensi pada dua mesin terpisah), yang hanya pernah berbicara satu sama lain lewat koneksi jaringan.



Gambar 14.1 — Peta jalan 16-kuliah OOP. Bab ini adalah Kuliah 14: membuat Pustaka berbicara dengan dunia luar lewat koneksi socket sungguhan.

14.2 Model Client-Server

Server adalah program yang dimulai lebih dulu, membuka socket yang terikat pada port tetap, lalu menunggu: ia tidak tahu sebelumnya siapa yang akan terhubung, atau berapa banyak klien yang akan ada. Klien adalah program yang sudah tahu alamat dan port server, dan memulai koneksinya. Begitu koneksi terbentuk, sebuah objek Socket ada pada KEDUA ujung -- server memegang satu Socket per klien yang terhubung, dan klien memegang satu Socket untuk koneksinya sendiri -- dan setiap sisi bisa membaca dari serta menulis ke stream socket itu persis seperti Bab 6 membaca dari dan menulis ke stream sebuah berkas (Gambar 14.2).



Gambar 14.2 — LibraryServer mendengarkan pada port tetap; LibraryClient setiap patron membuka koneksinya sendiri dan mendapat thread khususnya sendiri, tapi setiap thread membaca dari objek Library bersama yang sama.

> **Socket adalah jalan dua arah, bukan pipa satu arah**
>
> Objek Socket yang sama yang dipakai klien untuk MENGIRIM baris permintaan juga menjadi objek tempat ia MEMBACA respons server -- getOutputStream() dan getInputStream() adalah dua pandangan pada satu koneksi yang mendasarinya, bukan dua koneksi terpisah.

14.3 Membuka Socket yang Mendengarkan: ServerSocket

LibraryServer.java -- membuka socket dan menerima koneksi (diringkas)

```
try (ServerSocket serverSocket = new ServerSocket(PORT)) {
    System.out.println("LibraryServer listening on port " + PORT + "...");
    while (true) {
        Socket client = serverSocket.accept();   // memblokir sampai patron
terhubung
        Thread handlerThread = new Thread(new ClientHandler(client, library));
        handlerThread.start();
    }
}
```

accept() memblokir -- dan loop ini tidak pernah berakhir dengan sendirinya

serverSocket.accept() tidak kembali sampai seorang klien benar-benar terhubung; thread pemanggil hanya menunggu. Loop while (true) ini sengaja dan sesuai bagaimana server nyata berperilaku: ia terus menerima koneksi baru selama proses itu berjalan, dan biasanya dihentikan dari luar (Ctrl+C, atau process manager), bukan dari dalam kodenya sendiri.

14.4 Menangani Satu Klien: Protokol Berbasis Baris

Protokol Pustaka lewat kabel sengaja dibuat sederhana: klien mengirim tepat satu baris perintah, dan server mengirim kembali satu blok respons. FIND <judul> mencari satu item dan menjawab dengan satu baris OK|... atau ERROR|.... LIST menjawab dengan satu baris ITEM|... per item di perpustakaan, diikuti baris sentinel END sehingga klien tahu bloknya selesai. QUIT mengakhiri percakapan dengan baris BYE.

ClientHandler.java -- satu thread khusus per koneksi (diringkas)

```
public class ClientHandler implements Runnable {
    private final Socket socket;
    private final Library library;
    // konstruktor dihilangkan
    @Override
    public void run() {
        try (Socket s = socket;
             BufferedReader in = new BufferedReader(new
InputStreamReader(s.getInputStream()));
             PrintWriter out = new PrintWriter(s.getOutputStream(), true)) {
            String request;
            while ((request = in.readLine()) != null) {
                if (request.equalsIgnoreCase("LIST")) {
                    for (LibraryItem item : library.getAllItems()) {
                        out.println("ITEM|" + item.describe());
                    }
                    out.println("END");
                } // cabang FIND dan QUIT dihilangkan -- lihat
Code/Java/Chapter14/
            }
        } catch (IOException e) {
            System.out.println("Connection error: " + e.getMessage());
        }
    }
}
```

try-with-resources pada Socket juga menutup stream-nya

Membungkus Socket itu sendiri dalam try-with-resources (bersama BufferedReader dan PrintWriter yang dibangun dari stream-nya) berarti klien yang terputus di tengah percakapan, atau IOException apa pun, tetap menutup ketiganya dengan bersih -- disiplin persis yang diajarkan Bab 6 untuk berkas, diterapkan di sini pada koneksi jaringan.

14.5 Satu Thread Per Klien: Networking Bertemu Bab 13

LibraryServer menyerahkan setiap Socket yang diterima ke Thread yang benar-benar baru yang menjalankan ClientHandler, lalu segera kembali ke accept() untuk koneksi BERIKUTNYA -- ia tidak pernah menunggu percakapan satu klien selesai sebelum menerima yang lain. ini bukan materi baru; ini adalah Thread Bab 13, diterapkan pada jenis pekerjaan konkuren yang sungguh baru. Yang BARU di sini adalah state bersama yang disentuh setiap thread itu: objek Library yang sama, dibaca (tidak pernah ditulis) dari setiap ClientHandler sekaligus. Ringkasan Bab pada Bagian 14.9 menjelaskan secara eksplisit mengapa pola khusus itu -- banyak pembaca, tanpa penulis, selagi server berjalan -- aman tanpa mesin synchronized atau Lock Bab 13 sama sekali, dan persis apa yang harus berubah agar itu berhenti benar (Gambar 14.3).

Socket, Dibandingkan: Dua Cara Menyatakan Hal yang Sama

Kedua bahasa mengekspos konsep socket TCP yang sama di baliknya -- nama kelas dan urutan pemanggilan persisnya berbeda.

	Java	Python
Membuka socket yang mendengarkan	new ServerSocket(port)	socket.socket(...) lalu .bind((host, port))
Menerima SATU koneksi	serverSocket.accept() mengembalikan Socket	server_socket.accept() mengembalikan (conn, addr)
Membaca/menulis teks lewatnya	BufferedReader / PrintWriter membungkus stream milik Socket	conn.makefile("r"/"w") membungkus socket mentah
Satu thread per klien	new Thread(new ClientHandler(...)) .start()	threading.Thread(target=handle_client, args=...).start()

Tak satu pun bahasa membuat koneksi socket andal dengan sendirinya -- keduanya tetap butuh disiplin try-with-resources / context-manager yang sudah diajarkan Bab 6 dan Bab 13, sehingga koneksi yang putus atau klien yang lenyap di tengah permintaan tetap menutup setiap stream dan socket dengan bersih, bukan membocorkannya.

Gambar 14.3 — API socket dibandingkan lintas kedua jalur yang dipakai mata kuliah ini.

State bersama hanya-baca tidak otomatis aman-thread selamanya -- hanya selama ia tetap hanya-baca

Setiap ClientHandler di sini hanya memanggil getAllItems() dan findByTitle() -- keduanya hanya MEMBACA field Library. Jika versi mendatang protokol ini pernah menambahkan perintah BORROW yang memanggil addItem() atau removeItem() dari dalam ClientHandler, seluruh pelajaran Bab 13 akan berlaku lagi segera: banyak thread yang memutasi List dan Map yang sama tanpa sinkronisasi akan berisiko kehilangan update persis seperti yang didemonstrasikan BorrowCounter.

14.6 Sisi Klien: LibraryClient

LibraryClient.java -- mengirim satu perintah, membaca satu blok respons (diringkas)

```
try (Socket socket = new Socket(host, port);
     BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
     PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {
    out.println(command);   // autoFlush=true mengirim ini segera
    String line;
    while ((line = in.readLine()) != null) {
        System.out.println(line);
        if (line.equals("END") || line.startsWith("OK|")
            || line.startsWith("ERROR|") || line.equals("BYE")) {
            break;   // respons protokol sederhana ini selalu tepat satu blok
        }
    }
}
```

new Socket(host, port) adalah konstruktor sisi-KLIEN

Berbeda dari ServerSocket, yang menunggu koneksi secara pasif, memanggil new Socket(host, port) secara aktif MEMULAI upaya koneksi ke alamat itu -- ia berhasil (konstruktor kembali) atau melempar IOException (server tak terjangkau, menolak koneksi, atau tidak ada).

14.7 Mengompilasi dan Menjalankan: Server dan Klien, Dua Program Terpisah

Berbeda dari program driver tunggal setiap bab sebelumnya, demo bab ini benar-benar membutuhkan dua program berjalan PADA SAAT YANG SAMA: LibraryServer harus sudah berjalan sebelum LibraryClient bisa terhubung kepadanya.

```
$ javac Book.java DVD.java ItemNotFoundException.java Library.java
LibraryDemo.java \
      LibraryItem.java LibraryUtils.java Pair.java ReferenceBook.java
Renewable.java \
      ClientHandler.java LibraryServer.java LibraryClient.java
$ java LibraryServer &
LibraryServer listening on port 5051...
$ java LibraryClient localhost 5051 LIST
$ java LibraryClient localhost 5051 FIND Clean Code
```

14.8 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Arsitektur Profitina sendiri adalah sistem client-server pada skala yang jauh lebih besar: backend FastAPI-nya memainkan persis peran LibraryServer -- proses yang mendengarkan pada sebuah port dan menjawab permintaan dari banyak klien berbeda (browser, frontend-nya sendiri, tugas terjadwal) -- sementara setiap pemanggil itu memainkan peran LibraryClient, membuka koneksi, mengirim permintaan, dan membaca kembali respons. Protokol berbasis baris yang dirancang tangan di bab ini adalah, pada skala Profitina, HTTP dan JSON alih-alih FIND/LIST/QUIT -- tapi bentuk yang mendasarinya (alamat mendengarkan yang tetap, pertukaran request/response, satu handler per koneksi masuk) adalah pola persis yang diajarkan bab ini dari prinsip dasar.

14.9 Ringkasan Bab dan Poin-Poin Utama

- Server membuka ServerSocket yang terikat pada port tetap dan memanggil accept() dalam loop, yang memblokir sampai klien terhubung; klien memanggil new Socket(host, port) untuk secara aktif memulai koneksi.
- Kedua ujung koneksi yang terbentuk membaca dari dan menulis ke stream Socket yang SAMA -- getInputStream() dan getOutputStream() adalah dua pandangan pada satu koneksi dua-arah, bukan dua koneksi terpisah.
- Protokol bab ini sengaja dibuat sederhana: satu baris perintah masuk, satu blok respons keluar, dengan END eksplisit atau sentinel satu-baris sehingga klien tahu kapan respons selesai.
- LibraryServer menyerahkan setiap koneksi yang diterima ke Thread-nya sendiri -- alat yang sama yang diperkenalkan Bab 13, kini diterapkan pada I/O jaringan alih-alih kios yang disimulasikan -- sehingga banyak patron bisa dilayani pada saat yang sama.
- Setiap ClientHandler di sini hanya MEMBACA Library bersama; itulah persis mengapa belum diperlukan synchronized atau Lock -- begitu handler mana pun mulai menulis ke state bersama, seluruh pelajaran Bab 13 berlaku lagi.

Server yang hanya pernah bisa berbicara dengan satu klien pada satu waktu sebenarnya bukan server -- ia adalah program yang kebetulan menerima satu koneksi sekali.

14.10 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Jelaskan, dengan kata-kata Anda sendiri, mengapa serverSocket.accept() harus dipanggil lagi segera di dalam loop while (true), alih-alih hanya sekali sebelum loop dimulai.
2. Misalkan ClientHandler.run() TIDAK membungkus Socket itu sendiri dalam try-with-resources, hanya BufferedReader dan PrintWriter. Apa yang bisa salah jika klien terputus tiba-tiba di tengah respons LIST?
3. Protokol bab ini tidak punya cara bagi klien untuk memberi tahu server 'saya masih terhubung tapi tidak punya apa pun untuk dikatakan sekarang.' Jelaskan apa yang akan terjadi, pada sisi server, jika seorang klien terhubung lalu sama sekali tidak pernah mengirim baris apa pun.
4. Bagian 14.5 menjelaskan bahwa setiap ClientHandler hanya membaca Library, tidak pernah menulis kepadanya. Rancang (dengan kata-kata, bukan kode) sebuah perintah BORROW

yang AKAN perlu menulis ke Library, dan jelaskan secara spesifik alat Bab 13 mana yang akan Anda gunakan agar aman.

5. LibraryClient keluar setelah menerima tepat satu blok respons. Apakah membiarkan satu koneksi LibraryClient mengirim BANYAK perintah, satu demi satu, sebelum terputus, adalah perubahan sisi-klien atau sisi-server? Jelaskan kode sisi mana yang perlu berubah dan mengapa.

Lampiran 14.A — Log Verifikasi Eksekusi

Berkas Book.java, DVD.java, ItemNotFoundException.java, Library.java, LibraryItem.java, LibraryUtils.java, Pair.java, ReferenceBook.java, Renewable.java, ClientHandler.java, LibraryServer.java, dan LibraryClient.java yang lengkap (Code/Java/Chapter14/, disediakan bersama buku ini) dikompilasi dan dieksekusi pada OpenJDK sebelum bab ini ditulis -- dengan LibraryServer berjalan sebagai proses yang benar-benar terpisah, dan beberapa pemanggilan LibraryClient terhubung kepadanya lewat TCP/IP loopback nyata (bukan simulasi), termasuk dua klien yang terhubung PADA SAAT YANG SAMA untuk memastikan desain satu-thread-per-koneksi benar-benar melayani patron konkuren dengan benar. Output direproduksi verbatim di bawah ini dari satu eksekusi nyata.

```
$ java LibraryServer &
LibraryServer listening on port 5051...

$ java LibraryClient localhost 5051 LIST
ITEM|"Clean Code" [ISBN 9780132350884] - on the shelf
ITEM|"The Imitation Game" [ISBN 9900000000001] - on the shelf
ITEM|"Encyclopedia of Computer Science" [ISBN 9780123456789] - on the shelf
END

$ java LibraryClient localhost 5051 FIND Clean Code
OK|"Clean Code" [ISBN 9780132350884] - on the shelf

$ java LibraryClient localhost 5051 FIND Nonexistent Book
ERROR|Not found: Nonexistent Book

$ java LibraryClient localhost 5051 QUIT
BYE
```

Dua klien yang terhubung pada instan yang sama sama-sama dilayani dengan benar

Pengujian bab ini sendiri juga meluncurkan dua proses LibraryClient pada momen yang pada dasarnya sama (satu meminta FIND Clean Code, yang lain FIND The Imitation Game) terhadap satu LibraryServer yang sedang berjalan; keduanya menerima respons yang benar, memastikan desain satu-thread-per-koneksi benar-benar melayani patron konkuren secara independen, bukan sekadar satu per satu dengan penyamaran.

Referensi

[1] Oracle Corporation. "All About Sockets." The Java Tutorials, Custom Networking lesson. https://docs.oracle.com/javase/tutorial/networking/sockets/ (diakses Agustus 2026).

[2] Oracle Corporation. "Reading from and Writing to a Socket." The Java Tutorials, Custom Networking lesson. https://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html (diakses Agustus 2026).

[3] Oracle Corporation. "Thread" dan "Runnable." Java SE 21 API Documentation, diakses Agustus 2026 (referensi silang Bab 13).

BAB 15

Design Pattern: Menamai Apa yang Sudah Dilakukan Pustaka

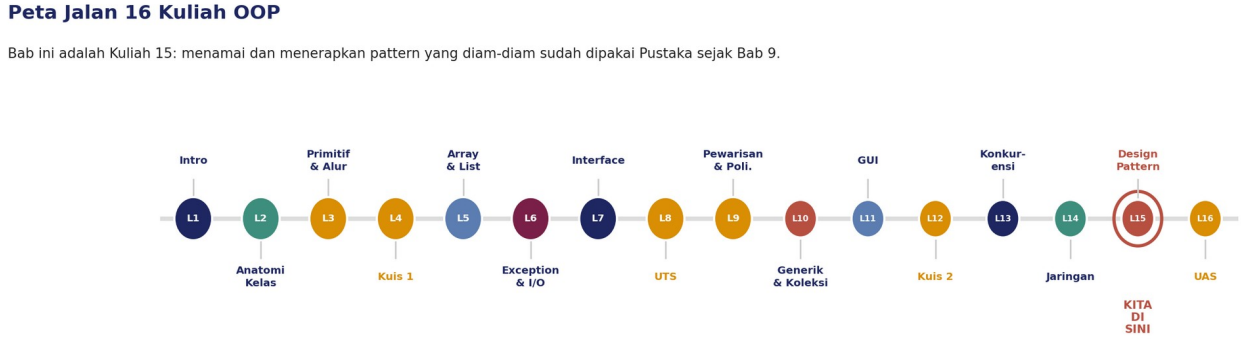
Setiap bab sejauh ini diam-diam meraih sebuah bentuk yang dapat dipakai ulang: rekonstruksi bertanda-tipe di Bab 9/10, View yang menggambar ulang dirinya sendiri ketika data Model berubah di Bab 11. Bab ini memberi tiga bentuk berulang itu namanya -- Factory Method, Observer, Singleton -- menerapkan dua di antaranya sebagai refactor Library sendiri yang nyata dan diungkapkan secara terbuka, dan memakai yang ketiga sebagai kisah peringatan tentang bahaya konkurensi yang sudah pernah ditemui mata kuliah ini, dalam penyamaran baru (Gambar 15.1).

Tujuan Pembelajaran — setelah bab ini Anda akan mampu:

(1) Menjelaskan apa itu design pattern, dan mengapa menamai sebuah bentuk struktural yang berulang memudahkan Anda mengenali, mendiskusikan, dan memakainya ulang secara sengaja. (2) Menerapkan pattern Factory Method untuk memusatkan logika rekonstruksi objek yang sebelumnya tersebar secara inline. (3) Menerapkan pattern Observer agar satu objek dapat mengumumkan perubahan state ke objek lain tanpa bergantung pada tipe konkretnya. (4) Menjelaskan mengapa inisialisasi lazy pada pattern Singleton naif adalah race condition yang nyata, dan mengapa inisialisasi eager menghilangkannya sepenuhnya, bukan sekadar mempersempitnya. (5) Membaca log eksekusi nyata yang mendemonstrasikan refactor design pattern yang memperbaiki bug yang sudah ada sebelumnya, dan log kedua yang mendemonstrasikan race condition yang direproduksi sesuai permintaan.

15.1 Rekap: Pustaka Sejauh Ini

Bab 13 memberi kode terkait-Library eksekusi konkuren yang aman; Bab 14 membuat Pustaka berbicara dengan dunia luar lewat koneksi socket nyata. Kedua bab itu, dengan caranya masing-masing, sudah memakai design pattern tanpa berhenti untuk menamainya: dispatch ClientHandler-per-koneksi Bab 14 adalah bentuk mirip-Command, dan pemisahan Model-View Bab 11 adalah relasi Observer buku teks dalam segala hal kecuali namanya. Bab ini berhenti meminjam pattern secara implisit dan mulai menerapkannya secara eksplisit -- pada Library yang SAMA yang telah dibangun mata kuliah ini sejak Bab 9, bukan pada contoh mainan baru.



Gambar 15.1 — Peta jalan 16-kuliah OOP. Bab ini adalah Kuliah 15: menamai dan menerapkan pattern yang diam-diam sudah dipakai Pustaka sejak Bab 9.

15.2 Mengapa Design Pattern?

Design pattern bukan library, framework, atau fitur bahasa -- ia adalah BENTUK bernama dan dapat dipakai ulang untuk menyelesaikan masalah desain yang berulang, tidak bergantung pada basis kode tertentu mana pun. "Factory Method" menamai bentuk "pusatkan kelas konkret mana yang dibangun, di satu tempat, alih-alih menyebarkan keputusan itu ke setiap pemanggil." "Observer" menamai bentuk "biarkan satu objek mengumumkan bahwa sesuatu berubah, tanpa objek itu perlu tahu siapa, jika ada, yang mendengarkan." Tak satu pun nama mengubah apa yang dilakukan kode; keduanya mengubah seberapa cepat dua developer bisa sepakat tentang APA sebuah potongan kode itu, begitu keduanya tahu nama pattern-nya. Bab ini secara sengaja menerapkan kedua pattern pada logika yang sudah dimiliki Library -- rekonstruksi tanda-tipe inline Bab 9/10, dan kebutuhan penyegaran-GUI implisit Bab 11 -- khusus agar "sebelum" dan "sesudah"-nya bisa dibandingkan secara langsung, alih-alih memperkenalkan pattern bersamaan dengan fungsionalitas yang benar-benar baru di mana perbandingannya akan lebih sulit dilihat.

> **Design pattern lebih sering dikenali belakangan daripada direncanakan lebih dulu**
>
> Katalog asli Gang of Four tahun 1994 sendiri disusun dengan mengamati bentuk-bentuk yang terus berulang di sistem berorientasi objek yang sudah bekerja, bukan dengan menciptakan bentuk lebih dulu lalu mencari penggunaannya belakangan. Pemisahan MVC Bab 11 sudah ada selama tiga bab sebelum bab ini pernah memakai kata "Observer" -- menamainya sekarang tidak mengubah kodenya; itu mengubah kosakata apa yang Anda miliki untuk mendiskusikannya.

15.3 Factory Method: Memusatkan "Kelas Mana yang Dibangun"

Sejak Bab 9, Library.loadFromFile() membaca baris bertanda-tipe (BOOK|..., REFBOOK|..., DVD|...) dan merekonstruksi subtipe LibraryItem konkret yang cocok, secara inline, di dalam method privat fromLine(). Keputusan itu -- "diberikan tanda ini, kelas mana yang saya bangun?" -- persis yang dinamai pattern Factory Method: sebuah method (atau kelas) khusus yang seluruh tugasnya adalah membangun objek dari tipe yang dipilih saat runtime, sehingga pemanggil LAIN mana pun yang kelak butuh keputusan yang sama bisa memakai ulang, bukan mengimplementasikan ulang tag-switch yang sama.

LibraryItemFactory.java -- logika rekonstruksi, diekstrak (lengkap)

```
public class LibraryItemFactory {
    private LibraryItemFactory() {
        // tak pernah diinstansiasi -- kelas utilitas tanpa-state, bentuk sama
seperti LibraryUtils
    }

    public static LibraryItem createFromLine(String line) {
        String[] parts = line.split("\\|");
        String tag = parts[0];
        switch (tag) {
            case "BOOK":
                return new Book(parts[1], parts[2], parts[3],
Integer.parseInt(parts[4]));
            case "REFBOOK":
                return new ReferenceBook(parts[1], parts[2], parts[3],
Integer.parseInt(parts[4]));
            case "DVD":
                return new DVD(parts[1], parts[2], Integer.parseInt(parts[3]));
            default:
                throw new IllegalArgumentException("Unknown item type tag: " +
tag);
        }
    }
}
```

Ini adalah REFACTOR dari perilaku yang sudah ada, bukan fitur baru: logika tag-switch itu sendiri tidak berubah dari Bab 9/10, hanya lokasinya yang berpindah. toLine() -- serialisasi, arah sebaliknya -- sengaja tetap di dalam Library, karena Factory Method secara spesifik tentang PEMBUATAN; menulis kembali objek yang sudah ada ke sebuah baris adalah tanggung jawab yang berbeda, dan prinsip bab ini sendiri (satu kelas, satu tanggung jawab yang jelas) menentang penggabungan keduanya hanya karena mereka adalah kebalikan satu sama lain.

Bug nyata yang terungkap oleh refactor ini: findByIsbn() setelah reload

Mengekstrak createFromLine() mengharuskan penulisan ulang loadFromFile() untuk memanggilnya, dan penulisan ulang loadFromFile() mengungkap cacat nyata yang sebelumnya tidak terdeteksi, ada tanpa berubah sejak Bab 9/10: loadFromFile() LAMA memanggil items.add(fromLine(line)) secara langsung -- mengisi daftar items, tapi tak pernah map byIsbn. Setiap item yang pernah dimuat ulang dari berkas tersimpan karena itu tak terlihat oleh findByIsbn(), meski findByTitle() bekerja dengan benar (menyembunyikan bug itu di setiap demo bab sebelumnya, yang tak satu pun kebetulan memanggil findByIsbn pada item yang dimuat ulang). loadFromFile() BARU kini menyalurkan setiap item yang direkonstruksi lewat addItem() -- dan addItem() sudah mengisi KEDUA koleksi. Lihat Lampiran 15.A untuk verifikasi sebelum/sesudah yang nyata atas perbaikan ini.

15.4 Observer: Mengumumkan Perubahan Tanpa Tahu Siapa yang Mendengarkan

GUI Bab 11 perlu menyegarkan daftar item yang ditampilkannya setiap kali koleksi Library yang mendasarinya berubah -- item ditambahkan atau dihapus -- tanpa Library pernah mengimpor apa pun

yang terkait-GUI; Library tidak boleh tahu Swing, atau tkinter, atau teknologi presentasi spesifik mana pun ada. Pattern Observer persis ini: Library memegang daftar pendengar yang memenuhi antarmuka yang sempit, dan memanggil method notifikasi masing-masing setelah setiap perubahan berhasil, membiarkan setiap pendengar bebas melakukan apa pun yang diinginkannya dengan notifikasi itu -- mencatatnya ke konsol, menggambar ulang jendela, atau mengabaikannya sepenuhnya.

LibraryObserver.java -- kontrak notifikasi (lengkap)

```
public interface LibraryObserver {
    void onItemAdded(LibraryItem item);
    void onItemRemoved(LibraryItem item);
}
```

Library.java -- mendaftarkan observer dan memberitahu mereka (diringkas)

```
private final List<LibraryObserver> observers = new ArrayList<>();

public void addObserver(LibraryObserver observer) {
    observers.add(observer);
}

private void notifyItemAdded(LibraryItem item) {
    for (LibraryObserver observer : observers) {
        observer.onItemAdded(item);
    }
}

public void addItem(LibraryItem item) {
    items.add(item);
    byIsbn.put(item.getIsbn(), item);
    notifyItemAdded(item);   // BARU pada bab ini
}
```

Karena loadFromFile() kini menyalurkan setiap item yang direkonstruksi lewat addItem() (Bagian 15.3), observer yang terdaftar dengan benar terpicu juga untuk item yang dimuat dari disk, tidak hanya item yang ditambahkan secara interaktif -- manfaat sampingan dari memperbaiki bug findByIsbn dengan cara yang sama. Demo bab ini sendiri hanya mendaftarkan satu observer konkret, ConsoleLogObserver, yang sekadar mencetak baris ke konsol; presenter GUI di masa depan bisa mendaftarkan observer kedua yang menggambar ulang Listbox sebagai gantinya, dan kode Library sendiri tidak perlu berubah sama sekali untuk mendukungnya.

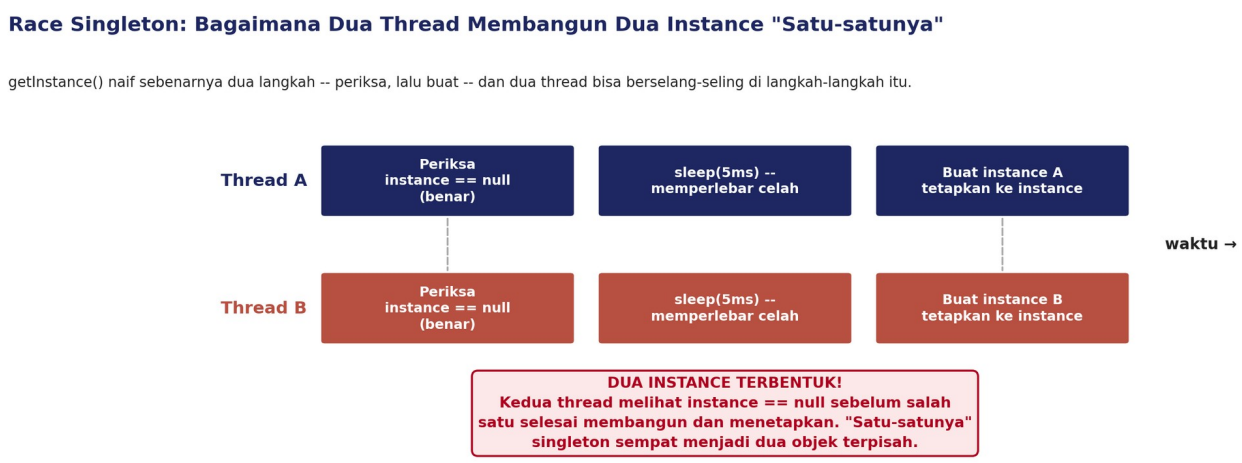
> **Observer adalah persis yang membuat desain Library terbuka untuk Bab 11 tanpa Bab 11 mengubah Library**
>
> Ini adalah inti dari pattern-nya: LibraryObserver didefinisikan dalam istilah yang sudah dipahami Library (sebuah LibraryItem), sehingga Library bisa bergantung padanya tanpa bergantung pada BAGAIMANA observer tertentu bereaksi. Menambahkan jenis observer kesepuluh nanti tidak memerlukan perubahan apa pun pada addItem(), removeItem(), atau loadFromFile() -- hanya kelas baru yang mengimplementasikan LibraryObserver dan satu pemanggilan addObserver().

15.5 Singleton: Kisah Peringatan

Singleton menjanjikan "tepat satu instance kelas ini pernah ada, dan setiap orang yang memintanya mendapat yang sama." Implementasi naif dan lazy muncul di tak terhitung banyak tutorial: periksa apakah instance sudah ada, dan jika belum, buat. Urutan check-then-act itu, secara struktural, persis bahaya yang sama yang didemonstrasikan race lost-update Bab 13 -- kecuali di sini race-nya adalah atas PEMBUATAN objek, bukan increment sebuah counter (Gambar 15.2, 15.3 dan 15.4).

NaiveSingleton.java -- versi klasik tanpa sinkronisasi (lengkap)

```
public class NaiveSingleton {
    private static NaiveSingleton instance;

    private NaiveSingleton() { }

    public static NaiveSingleton getInstance() {
        if (instance == null) {              // (1) periksa
            instance = new NaiveSingleton();  // (2) buat dan tetapkan
        }
        return instance;
    }
}
```



Gambar 15.2 — Dua thread sama-sama mengamati instance == null sebelum salah satu selesai membangun dan menetapkan: "satu-satunya" singleton sempat menjadi dua objek terpisah.

Catatan kejujuran: race ini persis tidak muncul dengan sendirinya dalam pengujian buku ini sendiri

Dua puluh thread yang berlomba melalui getInstance() dengan konstruktor tak dimodifikasi dan tanpa jeda hanya menghasilkan SATU instance berbeda, setiap kali, di lima eksekusi terpisah -- overhead startup thread biasa saja rupanya cukup untuk menyerialisasi thread melewati celah check-then-act yang kecil di lingkungan ini. Ini kelas temuan yang sama yang diungkapkan Bab 13 untuk race Python-nya (loop increment biasa tidak konsisten kehilangan update tanpa jeda pelebar eksplisit). Sesuai disiplin kejujuran baku buku ini, temuan negatif itu diungkapkan di sini alih-alih diam-diam diakali: sumber NaiveSingleton.java sendiri membawa Thread.sleep(5) singkat di antara pemeriksaan dan pembuatan, dan program demo menambahkan starting gate CountDownLatch sehingga kedua puluh thread mencapai getInstance() sedekat mungkin secara bersamaan seperti yang bisa diatur JVM. Kedua perubahan MEMPERLEBAR bahaya nyata yang sudah ada agar dapat diamati sesuai permintaan -- mereka tidak mengarang bahaya yang belum ada di sana. Dengan kedua perubahan itu diterapkan, race-nya bereproduksi dengan andal di tiga eksekusi berulang: dua puluh instance NaiveSingleton berbeda, setiap kali. Lihat Lampiran 15.A untuk output nyata yang tertangkap.

SafeSingleton.java -- inisialisasi eager menghilangkan race sepenuhnya (lengkap)

```
public class SafeSingleton {
    private static final SafeSingleton INSTANCE = new SafeSingleton();

    private SafeSingleton() { }

    public static SafeSingleton getInstance() {
        return INSTANCE;
    }
}
```

Singleton, Dibandingkan: Naif vs. Aman

Kedua kelas mengekspos bentuk getInstance() yang sama -- yang berbeda hanya KAPAN instance tunggalnya dibangun.

	NaiveSingleton	SafeSingleton
Kapan instance dibangun	Malas (lazy) -- pertama kali getInstance() dipanggil (ada celah check-then-act)	Segera (eager) -- sekali, saat kelas itu sendiri dimuat/diimpor, sebelum thread mana pun memanggil getInstance()
Jendela race untuk dua thread	Ada -- keduanya bisa melihat "belum dibangun" sekaligus	Tidak ada -- tak ada pemeriksaan untuk diperlombakan sama sekali
Biaya jika tak pernah benar-benar dipakai	Tidak ada -- dibangun hanya saat pemakaian pertama	Satu instance tetap dibangun walau getInstance() tak pernah dipanggil

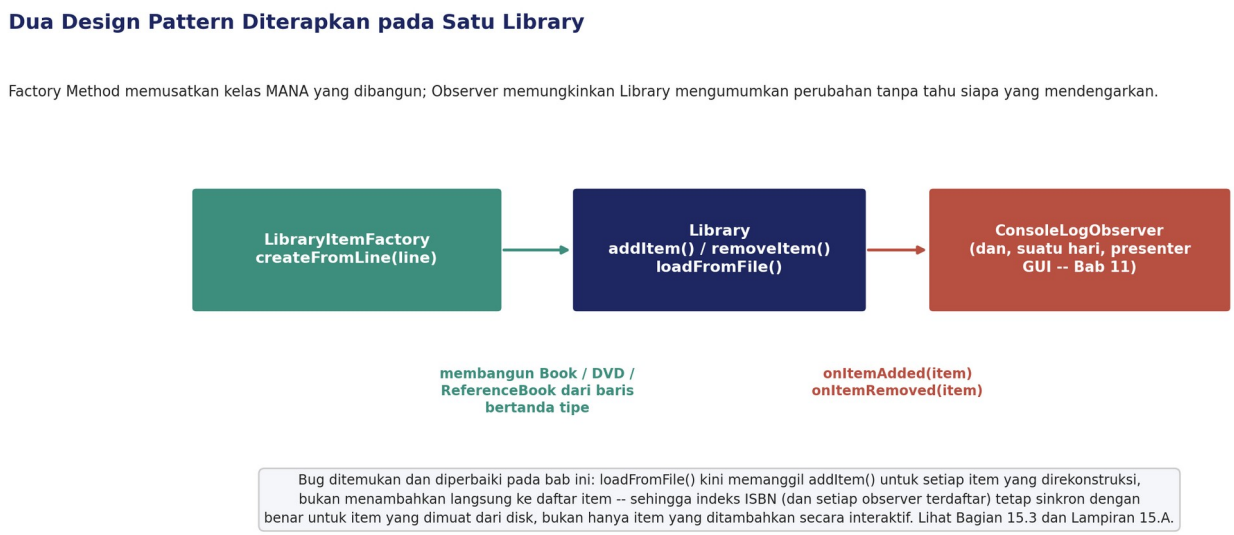
Ini kelas bahaya yang sama dengan race lost-update Bab 13 -- urutan check-then-act yang terlihat aman dibaca dari atas ke bawah, tapi tidak atomik. Perbaikannya di sini bersifat arsitektural (menghilangkan celahnya sama sekali), bukan menambal dengan synchronized/Lock.

Gambar 15.3 — NaiveSingleton vs. SafeSingleton: perbedaannya adalah KAPAN satu instance-nya dibangun, bukan bagaimana getInstance() terlihat dari luar.

Mengapa inisialisasi eager bebas-race, bukan sekadar kurang mungkin ber-race

JVM menjamin field statik sebuah kelas diinisialisasi tepat sekali, sebelum thread mana pun bisa mengamati kelas itu sama sekali -- sehingga INSTANCE sudah ada, sepenuhnya dibangun, sebelum getInstance() mungkin dipanggil dari lebih dari satu thread. Tak ada celah check-then-act di sini untuk diselang-selingi dua thread; ini adalah perbaikan arsitektural, bukan tambalan synchronized pada versi naif.

15.6 Design Pattern dan Library: Sebuah Studi Kasus



Gambar 15.4 — Factory Method dan Observer, keduanya diterapkan pada Library yang sama yang telah dibangun mata kuliah ini sejak Bab 9: LibraryItemFactory memusatkan pembuatan, Library memberi tahu observer terdaftar pada setiap perubahan.

PatternsDemo.java menjalankan kedua pattern yang diterapkan bersama-sama terhadap satu instance Library: tiga item ditambahkan (masing-masing memicu notifikasi Observer), satu dihapus (memicu notifikasi lain), library disimpan ke berkas dan dimuat ulang ke Library baru yang juga memiliki observer terdaftar (memicu notifikasi untuk setiap item yang dimuat dari disk, dan mendemonstrasikan rekonstruksi yang digerakkan Factory Method), dan akhirnya findByIsbn() dipanggil pada item yang dimuat ulang untuk mengonfirmasi langsung perbaikan bug Bagian 15.3. Setiap program demo Bab 9 hingga Bab 14 SENDIRI -- LibraryDemo.java khususnya -- dijalankan ulang tanpa modifikasi terhadap Library.java baru bab ini dan menghasilkan output identik seperti sebelumnya, memastikan kedua refactor bab ini mengubah struktur internal Library tanpa mengubah perilaku yang dapat diamati untuk pemanggil mana pun yang sudah ada (lihat Lampiran 15.A).

15.7 Mengompilasi dan Menjalankan

```
$ javac Book.java DVD.java ItemNotFoundException.java Library.java
LibraryDemo.java \
      LibraryItem.java LibraryUtils.java Pair.java ReferenceBook.java
Renewable.java \
      LibraryItemFactory.java LibraryObserver.java ConsoleLogObserver.java \
      NaiveSingleton.java SafeSingleton.java SingletonRaceDemo.java
PatternsDemo.java
$ java PatternsDemo
$ java SingletonRaceDemo
$ java LibraryDemo
```

15.8 OOP dalam Praktik: Profitina

> **Tentang kotak ini**
>
> Seperti di setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis -- menjelaskan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, tidak pernah logika bisnis atau kode sumber lengkap Profitina, yang tetap rahasia.

Basis kode Profitina sendiri memakai kedua pattern yang diterapkan bab ini secara nyata, pada skala produksi. Beberapa pipeline pemasukan-data-nya memusatkan "kelas parser mana yang menangani format sumber ini" di belakang satu fungsi factory, bentuk yang sama seperti LibraryItemFactory -- sehingga menambahkan dukungan untuk sumber data baru berarti menambahkan satu parser baru dan satu cabang factory baru, bukan memburu setiap titik pemanggilan yang dulu tahu formatnya secara langsung. Secara terpisah, pipeline event internal Profitina sendiri memungkinkan beberapa subsistem independen (logging, caching, notifikasi) bereaksi terhadap "sebuah laporan selesai dibuat" tanpa kode pembuatan-laporan mengimpor satu pun dari mereka secara langsung -- persis decoupling yang sama yang diberikan Observer kepada Library sehubungan dengan GUI Bab 11.

15.9 Ringkasan Bab dan Poin-Poin Utama

- Design pattern adalah bentuk bernama dan dapat dipakai ulang untuk masalah desain yang berulang -- menamainya tidak mengubah apa yang dilakukan kode, tapi membuat jauh lebih cepat bagi dua developer untuk sepakat tentang APA sebuah potongan kode itu.
- Factory Method memusatkan "kelas konkret mana yang saya bangun?" di satu tempat; bab ini menerapkannya dengan mengekstrak logika rekonstruksi tanda-tipe Library yang sudah ada ke LibraryItemFactory, refactor murni dari perilaku Bab 9/10.
- Refactor itu mengungkap dan memperbaiki bug nyata yang sebelumnya tidak terdeteksi: loadFromFile() tak pernah memperbarui indeks byIsbn, diam-diam merusak findByIsbn() untuk item mana pun yang pernah dimuat ulang dari disk sejak Bab 9/10.
- Observer memungkinkan Library mengumumkan perubahan addItem()/removeItem() ke pendengar terdaftar tanpa tahu atau peduli apa yang dilakukan pendengar itu -- persis decoupling yang akan dibutuhkan presenter GUI Bab 11 di masa depan.

- Inisialisasi lazy check-then-act Singleton naif adalah race condition nyata, secara struktural identik dalam bentuk dengan race lost-update Bab 13; inisialisasi eager menghilangkan bahayanya secara arsitektural alih-alih menyinkronkannya.

Menamai sebuah pattern tidak dengan sendirinya membuat kode lebih baik -- menerapkannya untuk menghilangkan penyebaran logika duplikat yang nyata, atau kopling nyata yang seharusnya tidak ada, itulah yang membuat kode lebih baik; namanya hanya cara Anda dan developer lain sepakat tentang perbaikan mana yang baru saja Anda buat.

15.10 Pertanyaan Review

Pertanyaan-pertanyaan ini untuk belajar mandiri dan diskusi kelas; tidak dinilai.

1. Bagian 15.3 menyatakan toLine() (serialisasi) tetap di dalam Library sementara createFromLine() (rekonstruksi) berpindah ke LibraryItemFactory. Jelaskan, dengan kata-kata Anda sendiri, mengapa kedua tanggung jawab ini TIDAK digabungkan ke kelas yang sama meski satu adalah kebalikan persis dari yang lain.
2. Jelaskan, secara tepat, mengapa bug yang dijelaskan Bagian 15.3 tak terlihat di setiap program demo dari Bab 9 hingga Bab 14, dan pemanggilan spesifik mana yang akan mengungkapnya di bab-bab sebelumnya itu.
3. Misalkan LibraryObserver konkret kedua ditambahkan yang melempar exception di dalam onItemAdded(). Berdasarkan implementasi notifyItemAdded() yang ditunjukkan Bagian 15.4, apa yang akan terjadi pada observer LAIN yang terdaftar, dan pada addItem() itu sendiri? Apakah ini desain yang baik, dan jika tidak, apa yang akan Anda ubah?
4. Bagian 15.5 mengungkapkan bahwa race Singleton naif tidak bereproduksi secara alami tanpa jeda tambahan dan starting gate. Jelaskan mengapa ini TIDAK berarti bahaya yang mendasarinya palsu atau aman diabaikan dalam kode nyata.
5. Rancang (dengan kata-kata, bukan kode) sebuah skenario di mana inisialisasi eager SafeSingleton akan menjadi pilihan yang benar-benar lebih buruk daripada inisialisasi lazy NaiveSingleton, meski ada race condition. Apa yang harus benar tentang konstruktor singleton itu agar trade-off itu penting?

Lampiran 15.A — Log Verifikasi Eksekusi

Berkas Book.java, DVD.java, ItemNotFoundException.java, Library.java, LibraryDemo.java, LibraryItem.java, LibraryUtils.java, Pair.java, ReferenceBook.java, Renewable.java, LibraryItemFactory.java, LibraryObserver.java, ConsoleLogObserver.java, NaiveSingleton.java, SafeSingleton.java, SingletonRaceDemo.java, dan PatternsDemo.java yang lengkap (Code/Java/Chapter15/, disediakan bersama buku ini) dikompilasi dan dieksekusi pada OpenJDK sebelum bab ini ditulis. Output direproduksi verbatim di bawah ini dari eksekusi nyata.

```
$ java PatternsDemo
--- Observer: notified on addItem() ---
[observer] added:   Clean Code
[observer] added:   The Imitation Game
[observer] added:   Encyclopedia of Computer Science

--- Observer: notified on removeItem() ---
[observer] removed: The Imitation Game

--- Factory Method: loadFromFile() reconstructs items via LibraryItemFactory ---
[observer] added:   Clean Code
[observer] added:   Encyclopedia of Computer Science

--- Bug found and fixed this chapter: findByIsbn() after reload ---
reloaded.size()                          -> 2
reloaded.findByIsbn("9780132350884")      -> Clean Code  (was null before this
chapter's refactor -- see Section 15.3)
```

Verifikasi sebelum/sesudah atas perbaikan bug findByIsbn

Pengujian sekali-pakai yang dikompilasi terhadap Library.java LAMA (pra-Bab-15) mengonfirmasi bug itu secara empiris: setelah putaran simpan/muat, findByIsbn("1111111111111") mengembalikan null sementara findByTitle("Test Book") dengan benar mengembalikan item. Pengujian yang identik dikompilasi terhadap Library.java BARU bab ini mengembalikan item yang benar dari findByIsbn() setelah putaran yang sama -- memastikan penulisan ulang loadFromFile() yang digerakkan Factory Method benar-benar memperbaiki cacat itu, bukan sekadar memindahkannya.

```
$ java SingletonRaceDemo
--- Demo 1: NaiveSingleton (unsynchronized lazy init) ---
Distinct NaiveSingleton instances observed: 20

--- Demo 2: SafeSingleton (eager init) ---
Distinct SafeSingleton instances observed:  1
```

Dua puluh thread, dua puluh instance NaiveSingleton berbeda -- setiap kali

Hasil persis ini (20 instance berbeda untuk NaiveSingleton, 1 untuk SafeSingleton) bereproduksi identik di tiga eksekusi berulang begitu pelebaran Thread.sleep(5) dan starting gate CountDownLatch yang dijelaskan Bagian 15.5 diterapkan -- memastikan race-nya nyata dan dapat diamati dengan andal, bukan kebetulan sekali jadi, begitu jendela check-then-act diberi ruang untuk dijalankan.

Referensi

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. (Katalog asli "Gang of Four" yang mendefinisikan Factory Method, Observer, Singleton, dan 20 pattern lainnya.)

[2] Oracle Corporation. "Understanding Class Members" dan urutan inisialisasi. Java SE 21 API Documentation / The Java Tutorials, diakses Agustus 2026 (dasar jaminan inisialisasi-eager SafeSingleton).

[3] Oracle Corporation. "Thread" dan "CountDownLatch." Java SE 21 API Documentation, diakses Agustus 2026 (referensi silang Bab 13; CountDownLatch dipakai starting gate SingletonRaceDemo bab ini).

BAB 16 · BAB LATIHAN

UAS: Proyek Capstone Client-Server Berbasis GUI dan Socket

Tidak ada materi baru di sini — satu proyek terintegrasi yang menggabungkan pemrograman GUI, konkurensi, jaringan, dan design pattern menjadi satu aplikasi client-server yang bekerja secara nyata.

Tujuan Pembelajaran — setelah menyelesaikan bab ini, Anda akan mampu:

(1) Memutuskan dan menentukan cakupan topik aplikasi client-server, serta membagi pekerjaan dalam tim kecil dengan spesifikasi dan deskripsi tugas masing-masing anggota. (2) Menghasilkan diagram komponen yang mencakup Server, Client, konektivitas basis data, GUI, dan testing/dokumentasi, serta menulis dokumentasi interface untuk setiap komponen dalam pengertian umum — bukan berarti harus berupa keyword interface Java. (3) Merancang dan mendeskripsikan protokol komunikasi antara client dan server, termasuk flowchart, diilustrasikan dengan contoh nyata. (4) Merancang test case dua arah, di mana setiap sisi pasangan client-server merancang pengujian untuk fungsionalitas sisi lainnya. (5) Membawa proyek melalui fase Spesifikasi, Desain, dan Implementasi, dengan dokumentasi level pengguna, kriteria performa, instalasi, dan troubleshooting. (6) Mendeteksi dan menangani server yang down atau client yang down secara baik pada kedua sisi, menggunakan exception handling, bukan membiarkan salah satu proses crash tanpa keterangan.

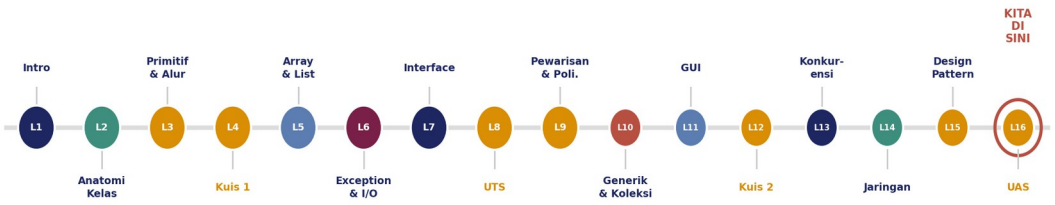
16.1 Rekap: Kuliah 13–15 secara Ringkas

Kuliah 1 hingga 11 membangun perangkat inti bahasa dan sudah direview secara penuh oleh Kuis 2 di Bab 12: pergeseran dari pemikiran prosedural ke berorientasi objek, anatomi kelas dan enkapsulasi, primitif dan alur kendali, Kuis 1, array dan List, exception dan file I/O, interface dan kelas abstrak, UTS, pewarisan dan polimorfisme, generik dan Collections Framework, serta pemrograman GUI dengan Model-View-Controller. Kuliah 13 membawa Library melampaui satu thread eksekusi, memperkenalkan Thread, synchronized, dan race lost-update yang dapat muncul dari urutan read-modify-write yang naif di bawah akses konkuren. Kuliah 14 membuka Library ke jaringan, membangun pasangan client-server di atas ServerSocket dan Socket, dengan protokol eksplisit dan diungkapkan secara jujur yang mengatur apa yang boleh dikatakan setiap sisi dan kapan. Kuliah 15 menamai dua bentuk yang sudah dimiliki Library — Factory Method dan Observer — dan menggunakan bentuk ketiga, Singleton, sebagai kisah peringatan tentang jenis race yang sama seperti Kuliah 13, kali ini pada konstruksi objek, bukan saldo rekening.

Bab final ini tidak menambahkan topik kelima yang baru. Sebagai gantinya, Anda diminta membangun satu aplikasi terintegrasi yang menggabungkan Kuliah 11, 13, 14, dan 15 sekaligus — sebuah sistem client-server berbasis GUI dan socket, dibangun dan dipertanggungjawabkan sebagai proyek capstone tim kecil (Gambar 16.1).

Peta Jalan 16 Kuliah OOP

Bab ini adalah Kuliah 16, UAS: proyek capstone yang menggabungkan Kuliah 11, 13, 14, dan 15 -- checkpoint terakhir mata kuliah ini.



Gambar 16.1 — Bab ini berada di Kuliah 16, UAS, di ujung peta jalan 16 kuliah OOP: sebuah capstone, bukan topik baru.

16.2 Cara Menggunakan Bab Latihan Ini

Apa bab ini — dan apa yang bukan

Ini adalah Bab Latihan: bab praktik, bukan bab materi baru. Berbeda dari Bab 4, 8, dan 12, proyek final ini bukan sekumpulan soal terpisah dan independen — ini adalah SATU aplikasi capstone terintegrasi, yang komponen-komponennya dapat ditelusuri kembali ke empat kuliah tertentu (lihat Gambar 16.2). Panduan di bawah ini disusun berdasarkan aspek (memilih topik, mendokumentasikan protokol, merancang pengujian, menangani kegagalan), masing-masing dengan petunjuk menuju praktik yang baik, bukan solusi lengkap yang sudah dikerjakan atau kode sumber referensi. Bab ini, seperti Bab 4, 8, dan 12 sebelumnya, tidak menyertakan subfolder code/ — inti dari sebuah capstone adalah Anda merancang dan membangunnya sendiri.

Asal Setiap Komponen Capstone

Proyek akhir ini adalah satu aplikasi terintegrasi -- setiap komponen di Bagian 16.3 dapat ditelusuri ke salah satu dari empat kuliah.

#	Komponen Capstone	Berasal dari
1	Lapisan GUI	K11 -- Model-View-Controller, komponen Swing/JavaFX berbasis event
2	Server Konkuren	K13 -- thread, sinkronisasi, menghindari race lost-update
3	Protokol Client-Server	K14 -- socket, stream, protokol komunikasi yang diungkapkan secara jujur
4	Desain yang Dapat Digunakan Ulang	K15 -- Factory Method, Observer, Singleton diterapkan pada kelas Anda sendiri

Gambar 16.2 — Keempat komponen proyek capstone ini masing-masing dapat ditelusuri kembali ke Kuliah 11, 13, 14, atau 15.

Sebelum Anda mulai

Baca Bagian 16.3 secara penuh — termasuk panduan penanganan kegagalan — sebelum menulis satu baris kode pun atau membentuk tim Anda. Aplikasi client-server yang dirancang tanpa protokol dan rencana kegagalan sejak awal jauh lebih sulit diperbaiki belakangan, persis seperti peringatan Kuliah 14.

16.3 Proyek Capstone Final

Bekerjalah dalam tim beranggotakan maksimal tiga mahasiswa. Bangun aplikasi client-server berbasis GUI dan socket pada topik apa pun yang benar-benar diminati tim Anda — sebuah game multipemain kecil dengan chat, sebuah klon media sosial yang ringan, sebuah alat kolaborasi real-time, sebuah portal informasi interaktif, atau, sesuai semangat asesmen asli yang menjadi model bab ini, "apa pun yang menurut Anda keren." Topiknya jauh lebih tidak penting dibandingkan apakah tim Anda dapat mendemonstrasikan setiap keterampilan di bawah ini di atasnya.

16.3.1 Memilih Topik Capstone Anda

Pilih topik yang dapat tim Anda deskripsikan dalam satu kalimat, dan yang jelas membutuhkan client, server, dan suatu bentuk state persisten atau bersama — topik di mana program konsol satu proses jelas merupakan alat yang salah. Tentukan judul proyek dan pitch satu paragraf sebelum melanjutkan ke desain komponen.

> **Petunjuk**
>
> Topik yang menyenangkan untuk dideskripsikan belum tentu cocok secara otomatis — periksa dahulu apakah topik itu secara alami membutuhkan setidaknya dua proses independen yang berkomunikasi lewat socket, setidaknya satu GUI, dan suatu perilaku yang dapat gagal lewat jaringan. Jika ide Anda bekerja baik-baik saja sebagai satu program konsol bergaya Library, itu belum menjadi topik client-server.

16.3.2 Komponen Proyek & Pembagian Kerja

Hasilkan diagram komponen yang mengidentifikasi, minimal: Server, Client, konektivitas basis data atau persistensi, lapisan GUI, dan testing/dokumentasi sebagai komponen tersendiri — bukan sebagai renungan belakangan. Untuk tim beranggotakan dua atau tiga orang, tetapkan kepemilikan yang jelas atas satu atau lebih komponen kepada setiap anggota, dan minta setiap anggota menulis spesifikasi singkat dan deskripsi tugasnya sendiri untuk bagian yang dimilikinya.

> **"Spesifikasi dan dokumentasi interface" adalah istilah umum di sini**
>
> Dokumentasi interface tidak selalu berarti keyword interface Java. Artinya: untuk setiap komponen yang bergantung padanya anggota tim lain atau bagian sistem lain, tuliskan apa yang diharapkannya sebagai input, apa yang dijaminkannya sebagai output, dan apa yang dijamin (atau tidak dijamin) ketika terjadi kegagalan — dalam bentuk apa pun (dokumen singkat, sekumpulan komentar Javadoc, sebuah tabel) yang mengomunikasikan hal ini dengan jelas kepada rekan tim yang belum membaca kode Anda.

16.3.3 Desain Protokol: Contoh "Ketuk-Ketuk"

Rancang dan deskripsikan, dengan flowchart, protokol komunikasi yang akan diikuti client dan server Anda — secara persis, sisi mana yang bicara lebih dulu, apa arti setiap pesan, dan bagaimana pertukaran itu berakhir. Deskripsi protokol yang hanya mendaftar jenis pesan, tanpa urutan, meninggalkan justru ambiguitas yang menyebabkan dua sisi yang dibangun secara independen mengalami deadlock atau salah memahami satu sama lain.

```
Server: "ketuk, ketuk"
Client: "siapa di sana?"
Server: "<seseorang>"
Client: "<seseorang> siapa?"
Server: "<sesuatu yang lucu>"
Server: "mau lagi?"
  -> Client menjawab "ya": pertukaran diulang dari awal
  -> Client menjawab "tidak": server menutup koneksi
```

Mengapa contoh sederhana ini penting

Setiap baris di atas tidak ambigu tentang giliran siapa yang berbicara dan apa arti sebuah jawaban -- persis properti yang dibutuhkan protokol Anda sendiri, apa pun topik Anda. Deskripsi protokol tanpa tingkat presisi ini belum menjadi protokol, hanya daftar nama pesan.

16.3.4 Desain Test Case Dua Arah

Karena client Anda menggunakan fungsionalitas server, dan server Anda, pada gilirannya, bergantung pada menerima permintaan yang terbentuk dengan baik dari client, desain test case harus berjalan dalam dua arah: anggota tim pemilik client sebaiknya merancang test case yang menguji perilaku server, dan anggota tim pemilik server sebaiknya merancang test case yang menguji perilaku client. Tidak ada sisi yang boleh hanya menguji kodenya sendiri secara terisolasi.

Rangkaian pengujian satu arah menyembunyikan bug nyata

Menguji hanya "apakah server saya merespons permintaan yang terbentuk dengan baik" tidak pernah menguji apa yang dilakukan server Anda terhadap permintaan yang cacat, pesan yang dikirim di luar urutan protokol, atau koneksi yang terputus di tengah pertukaran -- persis mode kegagalan yang diminta Bagian 16.3.6 untuk ditangani secara sengaja.

16.3.5 Fase Desain, Implementasi, dan Dokumentasi

Bawa proyek melalui tiga fase yang terlihat: Spesifikasi (apa yang harus dilakukan sistem, dan protokolnya), Desain (diagram komponen, tanggung jawab kelas, dan dokumentasi interface), dan Implementasi (kode yang benar-benar bekerja). Selain kode, hasilkan empat jenis dokumentasi: panduan level pengguna (cara menjalankan dan menggunakan aplikasi), pernyataan kriteria performa (apa arti "bekerja dengan benar" dan "bekerja dengan baik" untuk sistem Anda), panduan instalasi/cara menjalankan, dan panduan troubleshooting yang mencakup mode kegagalan di Bagian 16.3.6.

Rencana pengujian, jamak

Rencana pengujian Anda sebaiknya mendeskripsikan lebih dari satu bentuk pengujian: pengujian berbasis stub (client atau server palsu yang menggantikan yang asli), pengujian multithreaded (beberapa client mengakses satu server secara konkuren -- lihat Kuliah 13), dan, jika memungkinkan bagi setup tim Anda, pengujian terdistribusi di lebih dari satu mesin.

16.3.6 Menangani Kegagalan: Server Down, Client Down

Aplikasi client-server yang hanya bekerja saat kedua sisi sehat dan jaringan sempurna belumlah selesai. Client Anda harus mendeteksi dan merespons secara masuk akal ketika server tidak dapat dijangkau atau down di tengah sesi -- bukan hang tanpa henti atau crash dengan exception yang tidak

ditangani. Secara simetris, server Anda harus mendeteksi dan merespons secara masuk akal ketika sebuah client terputus secara tak terduga -- bukan membocorkan thread atau sumber daya socket, dan tidak membuat sisa server crash bagi client lain yang masih terhubung.

> **Ini persis peringatan Kuliah 14, pada skala proyek tim**
>
> Kuliah 14 membangun LibraryClient dan LibraryServer dengan try/catch eksplisit di sekitar setiap operasi socket, justru karena panggilan jaringan dapat gagal dengan cara yang tidak pernah dialami panggilan method lokal. Proyek capstone yang melewatkan ini mengulangi persis kesalahan yang disebutkan namanya di Bagian 14.6 kuliah tersebut.

16.4 Format UAS: Proyek Tim, Open-Book

UAS adalah proyek tim beranggotakan maksimal tiga mahasiswa, open-book, dinilai berdasarkan deliverable yang dideskripsikan di Bagian 16.3 secara keseluruhan: diagram komponen Anda, spesifikasi dan dokumentasi interface, deskripsi protokol dan flowchart, test case dua arah, dokumentasi per fase, dan aplikasi client-server yang bekerja dan dapat didemonstrasikan dengan penanganan kegagalan yang disengaja dan diungkapkan secara jujur.

Integritas akademik

Dengan mengumpulkan proyek ini, setiap anggota tim menegaskan bahwa pekerjaan yang dikumpulkan adalah upaya genuin tim mereka sendiri, bahwa kode, pustaka, atau referensi eksternal apa pun yang digunakan diungkapkan secara jujur dan bukan disajikan sebagai karya asli, dan bahwa kontribusi yang diklaim masing-masing anggota secara akurat mencerminkan pekerjaan yang benar-benar mereka lakukan.

Kriteria	Yang dinilai	Bobot
Desain	Diagram komponen, spesifikasi dan dokumentasi interface, serta kejelasan protokol/flowchart.	20%
Implementasi	Aplikasi client-server yang bekerja sesuai desain, termasuk lapisan GUI, konkurensi, dan jaringan.	50%
Analisis & evaluasi	Test case dua arah, rencana pengujian (berbasis stub, multithreaded, terdistribusi jika memungkinkan), dan perilaku penanganan kegagalan yang diungkapkan secara jujur.	25%
Kesimpulan	Ringkasan yang jelas dan jujur tentang apa yang dibangun tim, apa yang berhasil, dan apa yang akan dilakukan tim secara berbeda dengan waktu lebih banyak.	5%

16.5 OOP dalam Praktik: Profitina

Tentang kotak ini

Seperti pada setiap bab, kotak ini menghubungkan materi kuliah dengan Profitina, sebuah platform AI-visibility (Generative Engine Optimization) Indonesia yang nyata dan aktif, dibangun oleh penulis — mendeskripsikan DI MANA dan MENGAPA sebuah konsep digunakan dalam sistem nyata yang berjalan, bukan logika bisnis atau kode sumber lengkap Profitina yang bersifat rahasia.

Profitina sendiri, pada intinya, persis berbentuk seperti bab ini pada skala produksi: sebuah client yang menghadap GUI berkomunikasi lewat jaringan dengan sebuah server yang memegang logika bisnis dan data yang sesungguhnya, dibangun dari komponen dengan spesifikasinya sendiri, diuji secara dua arah, dan direkayasa sejak hari pertama untuk mendeteksi dan pulih dari dependensi yang down, bukan mengasumsikan jaringan selalu sempurna. Proyek capstone yang berhasil menerapkan keempat unsur bab ini dengan benar — GUI, konkurensi, jaringan, dan beberapa design pattern yang dinamai dengan baik — telah, dalam skala mini, membangun bentuk sistem yang sama dengan yang dibangun secara profesional oleh penulis mata kuliah ini.

16.6 Ringkasan Bab

- Bab ini tidak memperkenalkan materi baru — ini adalah capstone yang menggabungkan Kuliah 11 (GUI & MVC), Kuliah 13 (Konkurensi), Kuliah 14 (Jaringan), dan Kuliah 15 (Design Pattern) menjadi satu proyek terintegrasi.
- Tim beranggotakan maksimal tiga mahasiswa merancang dan membangun aplikasi client-server berbasis GUI dan socket, pada topik pilihan mereka, didemonstrasikan dengan diagram komponen, dokumentasi interface, deskripsi protokol dengan flowchart, dan test case dua arah.
- Dokumentasi mencakup empat jenis: level pengguna, kriteria performa, instalasi, dan troubleshooting — di samping fase Spesifikasi, Desain, dan Implementasi.
- Penanganan kegagalan dinilai, bukan opsional: baik server down maupun client down harus dideteksi dan ditangani secara baik, pada kedua sisi, menggunakan exception handling yang diungkapkan secara jujur.
- Bobot penilaian memberi bobot terberat pada Implementasi (50%), diikuti Analisis & Evaluasi (25%), Desain (20%), dan Kesimpulan (5%).

Enam belas kuliah lalu, mata kuliah ini dimulai dengan satu pertanyaan: apa yang berubah ketika sebuah program diorganisasikan di sekitar objek, bukan prosedur? Setiap bab sejak itu telah menjadi satu jawaban lagi atas pertanyaan tersebut, dalam potongan yang semakin besar — dan proyek final ini adalah kali pertama semua potongan itu harus bekerja bersama, sekaligus, saling berkomunikasi lewat jaringan, persis seperti perangkat lunak yang sesungguhnya.

Lampiran 16.A — Daftar Periksa Mandiri (Tidak Dinilai)

Sebelum tim Anda mengumpulkan tugas, jalankan seluruh proyek melalui daftar periksa ini. Butuh waktu satu sore dan menangkap sebagian besar hal yang paling sering ditandai penilai pada capstone semacam ini.

- Apakah diagram komponen mencakup Server, Client, konektivitas basis data/persistensi, GUI, dan testing/dokumentasi, dengan setiap komponen dimiliki oleh anggota tim yang disebutkan namanya?
- Apakah ada spesifikasi tertulis untuk setiap komponen yang bergantung padanya anggota tim lain — input, output, dan jaminan kegagalan — bukan sekadar komentar Javadoc pada sebuah signature method?
- Apakah deskripsi protokol menyertakan flowchart, dan apakah secara tidak ambigu menyatakan giliran siapa yang berbicara di setiap langkah, seperti contoh Ketuk-Ketuk?
- Apakah pemilik client merancang test case untuk server, dan apakah pemilik server merancang test case untuk client — bukan hanya pengujian yang masing-masing tulis untuk kodenya sendiri?
- Apakah rencana pengujian mencakup pengujian berbasis stub, pengujian multithreaded dengan lebih dari satu client konkuren, dan (jika memungkinkan) pengujian terdistribusi di lebih dari satu mesin?
- Apakah client mendeteksi dan menangani server yang down atau down di tengah sesi, tanpa hang atau crash tanpa keterangan?
- Apakah server mendeteksi dan menangani client yang terputus secara tak terduga, tanpa membocorkan sumber daya atau memengaruhi client lain yang masih terhubung?
- Apakah keempat deliverable dokumentasi (level pengguna, performa, instalasi, troubleshooting) semuanya ada, dan apakah rekan tim yang baru bergabung hari ini dapat menjalankan sistem hanya dari dokumentasi tersebut?

Referensi

[1] Format asesmen dan rubrik penilaian bab ini dimodelkan secara dekat dari UAS nyata mata kuliah ini (EF234302 Pemrograman Berorientasi Objek): proyek tim beranggotakan maksimal tiga mahasiswa, open-book, membangun aplikasi client-server berbasis GUI dan socket pada topik pilihan tim, dinilai berdasarkan Desain (20%), Implementasi (50%), Analisis & Evaluasi (25%), dan Kesimpulan (5%). Berbeda dari UTS yang asli (lihat Referensi [1] Bab 8), yang konten pohon rekursif dan berbasis list-nya berada di luar cakupan OOP dari rebuild ini dan membutuhkan soal pengganti yang dirancang baru, konten wajib UAS yang asli — diagram komponen yang mencakup Server/Client/GUI/konektivitas basis data/testing, spesifikasi dan dokumentasi interface, protokol komunikasi yang diungkapkan secara jujur diilustrasikan dengan contoh Ketuk-Ketuk yang sama seperti di atas, desain test case dua arah, dokumentasi per fase, dan penanganan kegagalan server-down/client-down yang disengaja — sesuai secara langsung dengan persis apa yang benar-benar diajarkan rebuild ini di Kuliah 11, 13, 14, dan 15, dan digunakan kembali di sini sebagian besar apa adanya. Rincian administratif dari dokumen asli (alamat email pengumpulan, tanggal, konvensi penamaan berkas, dan Ikrar Integritas

Akademiknya sendiri yang bernuansa religius) telah digeneralisasi atau diparafrasakan untuk audiens buku ajar, bukan direproduksi kata demi kata.

[2] Oracle Corporation. "All About Sockets." The Java Tutorials. https://docs.oracle.com/javase/tutorial/networking/sockets/ (diakses Agustus 2026).

[3] Oracle Corporation. "Lesson: Concurrency." The Java Tutorials. https://docs.oracle.com/javase/tutorial/essential/concurrency/ (diakses Agustus 2026).