Nama : Repi Meilani Putri
Nim : 4241250022
Kelas : PSIK 24A

UTS PBO

Essay

1. Jelaskan bagaimana prinsip encapsulation, inheritance, polymorphism, dan abstraction saling mendukung dalam membangun sistem perangkat lunak yang mudah dikembangkan dan dipelihara. Sertakan contoh analogi dalam kehidupan nyata untuk masing-masing konsep.

Jawab:

a. Encapsulation (Enkapsulasi)

* Prinsip : Enkapsulasi adalah pembungkusan data (atribut) dan metode (fungsi) yang beroperasi pada data tersebut dalam satu unit yang disebut objek. Ini juga melibatkan penyembunyian detail implementasi internal objek dari dunia luar, hanya mengekspos antarmuka yang diperlukan.

* Bagaimana mendukung pengembangan dan pemeliharaan :

- Modularity : objek menjadi unit yang mandiri. Perubahan pada implementasi internal suatu objek tidak akan mempengaruhi bagian lain dari sistem selama antar muka publiknya tidak berubah. Ini membuat kode lebih mudah untuk dipahami, diuji, dan dimodifikasi secara individual.
- Data Hiding (Penyembunyian Data) : Mencegah akses langsung dan tidak sah ke data internal objek, sehingga mengurangi risiko kesalahan atau korupsi data. Data hanya bisa dimanipulasi melalui metode yang telah ditentukan, memastikan integritas data.
- Reusability : Objek yang terenkapsulasi dengan baik lebih mudah untuk digunakan kembali di bagian lain dari aplikasi atau bahkan di aplikasi yang berbeda.

* Analogi kehidupan nyata :

Bayangkan sebuah mobil.

- Data : mesin, roda, kemudi, pedal gas, rem, dll.
- Metode : menyalakan mesin, menginjak gas, menginjak rem, memutar kemudi.
- Enkapsulasi : Anda tidak perlu tahu bagaimana persisnya mesin bekerja (detail internalnya tersembunyi), tetapi anda bisa mengendarainya menggunakan antar muka yang disediakan (kemudi, pedal, tombol). Jika pabrikan mengubah desain internal mesin, selama cara anda menggunakan kemudi dan pedal tidak berubah, anda bisa mengendarainya. Ini membuat mobil mudah dikemudikan dan dipelihara tanpa perlu menjadi mekanik.

b. Inheritance (Pewarisan)

* Prinsip : Pewarisan memungkinkan sebuah kelas baru (kelas anak / subkelas) untuk mewarisi atribut dan metode dari kelas yang sudah ada, (kelas induk / superkelas). Ini membentuk hubungan "adalah sebuah" (is-a).

* Bagaimana mendukung pengembangan dan pemeliharaan :

- Code Reusability : mengurangi duplikasi kode. Atribut dan metode yang umum dapat didefinisikan sekali di kelas induk dan digunakan kembali oleh semua kelas anak.
- Extensibility : memungkinkan penambahan fungsionalitas baru dengan membuat kelas anak tanpa mengubah kelas induk yang sudah ada. Ini sangat berguna ketika sistem perlu diadaptasi untuk memenuhi kebutuhan baru.
- Maintability : Perubahan pada fungsionalitas umum di kelas induk akan secara otomatis diterapkan pada semua kelas anak, mengurangi upaya pemeliharaan.

* Contoh Analogi kehidupan nyata :

Bayangkan kategori "kendaraan" sebagai kelas induk.

- Kelas induk (kendaraan) : memiliki atribut umum seperti kecepatan, jumlah roda, dan metode seperti "bergerak".
- Kelas Anak (mobil, sepeda motor, truk) : masing-masing mewarisi atribut dan metode dari "kendaraan".
  - Mobil (subkelas dari kendaraan) mewarisi 'kecepatan', 'jumlah rod', dan 'bergerak'. Ia juga memiliki atribut dan metode spesifiknya sendiri seperti 'jumlah pintu' atau 'menyalakan AC'.
  - Sepeda motor (subkelas dari kendaraan) juga mewarisi, tetapi memiliki 'jenis stang' dan 'memiringkan'. Ini mengurangi kebutuhan untuk mendefinisikan 'kecepatan' dan 'bergerak' berulang kali untuk setiap jenis kendaraan.

C. Polymorphism (Polimorfisme)

* Prinsip : Polimorfisme (banyak bentuk) memungkinkan objek dari kelas yang berbeda untuk diperlakukan sebagai objek dari kelas yang sama (kelas induknya). Ini paling sering diimplementasikan melalui method overridding (metode dengan nama yang sama di kelas anak memiliki implementasi yang berbeda) dan method overloading (metode dengan nama yang sama tetapi parameter yang berbeda).

* Bagaimana mendukung pengembangan dan pemeliharaan :

- Flexibility : kode menjadi lebih fleksibel karena dapat berinteraksi dengan berbagai jenis objek melalui antar muka.
- Extensibility : ketika kelas baru ditambahkan (melalui pewarisan), kode yang sudah ada tidak perlu dimodifikasi untuk mengakomodasi kelas baru tersebut, selama kelas baru tersebut mengimplementasikan antar muka yang sama.
- Simpler Code : mengurangi kompleksitas kode karena anda dapat menulis kode yang lebih generik yang bekerja dengan objek dari berbagai kelas, daripada menulis kode spesifik untuk setiap kelas

* Contoh Analogi kehidupan nyata :

Kembali ke analogi "kendaraan".

- Polimorfisme : Anda memiliki daftar berbagai jenis kendaraan (mobil, sepeda motor, truk). Anda dapat atau bisa memberikan perintah generik "bergerak" kepada setiap kendaraan dalam daftar tersebut.
  - Ketika anda menyuruh "mobil" untuk 'bergerak', ia mungkin melaju dengan empat roda.
  - Ketika anda menyuruh "sepeda motor" untuk 'bergerak', ia mungkin melaju dengan dua roda dan suara mesin yang berbeda.
  - Ketika anda menyuruh "truk" untuk 'bergerak', ia mungkin bergerak lebih lambat dengan beban berat. Meskipun metode yang dipanggil sama ('bergerak'), implementasi spesifiknya berbeda tergantung pada jenis objek (mobil, sepeda motor, truk), yang memanggilnya. Anda tidak perlu menulis kode terpisah untuk "menggerakkan mobil", "menggerakkan sepeda motor", dll.

d. Abstraction (Abstraksi)

* Prinsip : Abstraksi adalah proses menyembunyikan detail implementasi yang tidak penting dan hanya menampilkan fungsionalitas esensial kepada pengguna. Ini berfokus pada "apa yang dilakukan" dari pada "bagaimana itu dilakukan". Abstraksi ini sering dicapai melalui penggunaan kelas abstrak dan antarmuka.

* Bagaimana mendukung pengembangan dan pemeliharaan :

- Simplicity : menyederhanakan kompleksitas sistem dengan hanya mengekspos informasi yang relevan, sehingga pengguna (programmer lain) tidak perlu memahami detail internal yang rumit.
- Decoupling : memisahkan antarmuka dari implementasi. ini berarti perubahan pada implementasi internal tidak mempengaruhi kode yang menggunakan antar muka tersebut.
- Modularity : mirip dengan enkapsulasi, abstraksi membantu dalam menciptakan modul yang terdefinisi dengan baik dan mudah dipahami.

* Contoh Analogi kehidupan nyata :

Bayangkan remote control TV

- Abstraksi : kita hanya melihat tombol-tombol seperti "Power", "Volume up", "Channel down". Kita tidak perlu tahu bagaimana sirkuit internal remote control bekerja, bagaimana sinyal inframerah dihasilkan, atau bagaimana TV memproses sinyal tersebut.
- Fungsionalitas Esensial : yang penting bagi kita adalah menekan "Power" akan menyalakan TV, dan menekan "Volume up" akan menaikkan volume. Detail implementasi dibalik tombol-tombol tersebut disembunyikan. Jika pabrikan TV mengubah cara internal TV menyala, selama remote control tetap memiliki tombol "Power", kita masih bisa menggunakannya.

* Bagaimana mereka saling mendukung :

Keempat Prinsip ini tidak bekerja secara, melainkan saling mendukung untuk menciptakan Sistem yang kokoh :

- enkapsulasi adalah fondasi yang menyediakan unit dasar yang mandiri dan terlindungi. Ini membuat objek-objek menjadi "kotak hitam" yang bisa diandalkan.
- Pewarisan memanfaatkan enkapsulasi dengan memungkinkan kelas-kelas baru untuk membangun di atas kelas yang sudah ada tanpa perlu menduplikasi kode. Ini menciptakan hierarki yang terstruktur.
- Polimorfisme memanfaatkan Pewarisan dan enkapsulasi untuk memungkinkan interaksi yang fleksibel dengan objek dari berbagai jenis melalui antar muka umum. Ini memfasilitasi penulisan kode yang lebih generik dan dapat diperluas.
- Abstraksi bekerja pada tingkat yang lebih tinggi, menggunakan enkapsulasi untuk menyembunyikan detail dan menyajikan gambaran yang lebih sederhana, dan sering kali menggunakan polimorfisme (melalui antar muka atau kelas abstrak) untuk mendefinisikan kontrak yang harus di penuhi oleh implementasi konkret.

Dengan menerapkan prinsip-prinsip ini secara bersamaan, pengembang dapat membangun sistem perangkat lunak yang kompleks namun tetap terstruktur, mudah dipahami, dan diubah, diperluas seiring waktu.

2. Apa kelebihan menggunakan Java versi terbaru (Java 21) dibanding versi-versi sebelumnya dalam konteks pengembangan berbasis OOP? Berikan minimal dua fitur modern Java 21 dan jelaskan bagaimana fitur tersebut menyederhanakan pengembangan aplikasi OOP.

Jawab:

* Kelebihan menggunakan Java 21 dibanding versi sebelumnya dalam konteks OOP :
  - kode lebih ringkas dan ekspresif
  - Penanganan data yang lebih efisien dan aman
  - konkurensi yang lebih sederhana dan skalabel
  - Peningkatan Performa
* Fitur modern Java 21

a. Virtual Theards (JEP 444)

Virtual theards, bagian dari Project Loom, adalah implementasi lightweight thread (thread ringan) yang dikelola oleh Java Virtual Machine (JVM), bukan oleh sistem operasi. Berbeda dengan Platfrom threads tradisional yang langsung dipetakan oleh threads OS dan memiliki overhead memori yang signifikan, virtual threads sangat "murah" dalam hal memori dan CPU.

•Bagaimana menyederhanakan pengembangan aplikasi OOP :

Dalam pengembangan aplikasi OOP, seringkali berhadapan dengan kebutuhan konkurensi, di mana banyak objek atau komponen perlu melakukan tugas secara bersamaan. Tradisionalnya ini, melibatkan penggunaan Platfroms threads yang mahal, yang dapat menyebabkan masalah skalabilitas atau memaksa developer

menggunakan model pemograman asinkron / reaktif yang kompleks (seperti CompletableFuture, atau kerja reaktif seperti Reactor / Rx Java) untuk menghindari pemblokiran thread OS.

- Virtual Threads menyederhanakan ini dengan cara :
  - Model Pemograman imperatif yang familiar
  - Skalabilitas yang luar biasa

b. Record Patterns (JEP 440) dan Pattern Matching for switch (JEP 441)

- Record (dari Java 16) : Record adalah kelas khusus yang di desain untuk menjadi data carrier (pembawa data) yang immutable. mereka secara otomatis menyediakan konstruktor, accessor methods (seperti getX()), equals(), hashCode(), dan toString(). Ini mengurangi boiler plate yang signifikan untuk objek data sederhana.
- Record Patterns (dari Java 19, finalized in 21) : memungkinkan untuk "mendekonstruksi" (membongkar) sebuah objek record langsung dalam sebuah ekspresi instanceof atau dalam case label dari switch statement / expression. Ini memungkinkan untuk mengekstrak komponen-komponen record tersebut ke dalam variabel lokal secara langsung.
- Pattern matching for switch (dari Java 17, finalized in 21) : memperluas kemampuan switch statement / expression untuk bekerja dengan Patterns (Pola), termasuk type patterns dan record patterns. Ini memungkinkan untuk menulis logika penanganan tipe yang lebih ringkas dan aman.

- Bagaimana menyederhanakan Pengembangan aplikasi OOP :

Dalam OOP, seringkali kita perlu memproses objek berdasarkan tipenya atau berdasarkan strukturnya (terutama jika menggunakan konsep Sealed Classes atau hierarki kelas). Sebelum fitur ini, kita menggunakan serangkaian if-else if dengan instanceof dan kemudian casting eksplisit, yang rentan terhadap ClassCastException dan membuat kode bertele-tele.

- Record Patterns (JEP 440) dan pattern Matching for switch (JEP 441) menyederhanakan ini dengan cara :
  - Penanganan tipe yang lebih bersih dan aman
  - Dekomposisi objek data yang kompleks (Record Patterns)

Secara keseluruhan, Java 21, mendorong Pengembangan aplikasi OOP menjadi lebih bersih, ringkas, aman, dan skalabel, memungkinkan developer untuk menulis kode yang lebih fokus pada solusi masalah bisnis dan kurang pada detail implementasi yang rumit.

3.) Mahasiswa sering kali salah memahami perbedaan antara class dan object. Jelaskan secara detail perbedaan keduanya dan berikan contoh penggunaan class dan object dalam konteks program manajemen data mahasiswa.

Jawab :

a. Class (kelas)

(KKY)

Class adalah blue Print (cetak biru), template, atau rancangan untuk membuat objek. Ia mendefinisikan karakteristik (data) dan Perilaku (fungsi atau metode) yang akan dimiliki oleh objek-objek yang dibuat dari kelas tersebut. Class itu seperti desain atau spesifikasi sebuah entitas.

* Analogi :

Bayangkan Class seperti atau sebagai cetak biru atau denah rumah. Denah ini mendefinisikan bahwa Setiap rumah yang dibangun dari denah ini akan memiliki :

- Jumlah kamar tidur
- Jumlah kamar mandi
- Luas tanah
- Pintu masuk
- Jendela
- Dan memiliki aksi seperti : 'membuka pintu', 'menyalakan lampu'.

Denah itu sendiri bukanlah rumah yang bisa dihuni. Itu hanya konsep atau rencana.

* Karakteristik kunci Class :

- Tidak alokasi memori : Sebuah Class itu sendiri tidak menempati ruang memori (heap) saat Program dijalankan. Ia hanya ada di area memori khusus JVM yang menyimpan definisi kelas (mendatanya).
- Logis / konseptual : Ia adalah entitas logis atau konseptual.
- Didefinisikan Sekali : Biasanya mendefinisikan sebuah Class satu kali dalam kode
- Untuk membuat objek : Tujuan utamanya adalah membuat objek (instance).

b. Object (objek)

Object adalah instance (contoh nyata) dari sebuah Class. Ia adalah entitas konkret yang dibuat berdasarkan cetak biru (Class) dan memiliki nilai-nilai spesifik untuk karakteristik yang di definisikan oleh kelasnya. Object adalah sesuatu yang nyata yang bisa berinteraksi.

* Analogi :

Melanjutkan analogi rumah, jika Class adalah denah rumah, maka Object adalah rumah-rumah konkret yang sebenarnya dibangun berdasarkan denah itu.

- Rumah Pertama : 3 kamar tidur, 2 kamar mandi, luas 100 m^2, cat putih
- Rumah kedua : 4 kamar tidur, 3 kamar mandi, 150 m^2, cat biru.

Meskipun keduanya dibangun dari denah yang sama, mereka adalah entitas fisik yang terpisah dengan karakteristik (nilai) yang spesifik dan unik.

* Karakteristik kunci Object :

- Alokasi memori : Setiap object menempati ruang memori (heap) yang terpisah saat program dijalankan.
- Fisik / nyata : Ia adalah entitas fisik atau nyata dalam memori program.
- Bisa banyak : bisa membuat Object banyak dari satu Class yang sama.
- Memiliki State dan Behavior : Setiap Object memiliki State (nilai-nilai dari atributnya) dan behavior (metode yang bisa dipanggil padanya).

Perbedaan Utama :

Fitur	Class (kelas)	Object (Objek)
Sifat	Blueprint / Template / Desain	Intance / Realisasi / Entitas Nyata
Keberadaan	konseptual / Logis	Fisik / konkret (di memori)
Memori	Tidak menempati memori (hanya definisi)	Menempati memori (heap)
Jumlah	Hanya satu definisi per tipe	Bisa ada banyak objek dari satu kelas
Fokus	Mengidentifikasi struktur dan perilaku Potensial	Memiliki state (nilai data) dan behavior aktual
kata kunci	didefinisikan dengan Class	Dibuat dengan new

* Contoh Penggunaan Class dan Object dalam Program manajemen data mahasiswa

a. Peran Class (mahasiswa)

Pertama, para Pengembang sistem perlu mendefinisikan apa itu "Mahasiswa" dari sudut pandang Progam. Mereka akan membuat sebuah Class yang diberi nama Mahasiswa.

- Class Mahasiswa ini bukan mahasiswa beneran. Ini hanya seperti cetak biru atau rancangan teknis yang menjelaskan bahwa setiap mahasiswa dalam sistem akan memiliki :
  - Sebuah nomor induk Mahasiswa (Nim) yang unik.
  - Sebuah nama lengkap
  - Sebuah Jurusan atau Program Studi
  - Sebuah tahun angkatan
  - Sebuah IPK (Indeks Prestasi kumulatif)
- Selain itu, Class Mahasiswa juga akan mendefinisikan tindakan-tindakan yang bisa dilakukan oleh atau terhadap setiap mahasiswa. misalnya :
  - Menampilkan semua informasinya
  - Memperbarui nilai IPK-nya
  - Mengubah jurusannya.

Jadi, Class Mahasiswa adalah definisi abstrak yang menyatakan : "Beginilah cara kita akan mempresentasikan seorang mahasiswa di dalam program ini". Ini adalah sebuah konsep, bukan data aktual.

b. Peran Object (mahasiswa nyata)

Setelah Class Mahasiswa didefinisikan, barulah kita mulai "membuat" mahasiswa sungguhan di dalam Program. Setiap kali kita mendaftarkan siswa baru ke dalam sistem, kita akan membuat sebuah Object dari Class Mahasiswa.

- Mahasiswa Pertama (Object 1) : Budi santoso
  - Ketika Budi santoso mendaftar, sistem akan membuat object baru berdasarkan Class Mahasiswa.
  - Object ini akan memiliki nilai spesifik untuk atribut-atributnya : NIM "2022001", Nama "Budi Santoso" Jurusan "Teknik Informatika", Angkatan 2022, IPK 0.0 (awal).
  - Object "Budi Santoso" ini adalah entitas nyata di dalam memori program. kita bisa menanyanya untuk : "Budi, tolong tampilkan informasimu." atau, "Budi, IPK-mu sekarang 3,85."

- Mahasiswa kedua (Object 2) : Putri Amirah
  - Saat Putri Mahira mendaftar, sistem akan membuat object lain yang sama sekali terpisah dari object Budi, juga berdasarkan Class Mahasiswa.
  - Object ini akan memiliki nilai spesifiknya sendiri : NIM "2021005", Nama "Putri Amirah", Jurusan "Sistem Informasi", Angkatan 2021, IPK 0.0.
  - Kita bisa meminta nya untuk : "Putri, tampilkan informasimu". Ini tidak memengaruhi object Budi.
- Mahasiswa ketiga (Object 3) : Dion Mahesa, dan seterusnya.

Setiap kali kita mendaftar mahasiswa baru, kita sebenarnya menciptakan object baru yang merupakan perwujudan unik dari Class Mahasiswa, masing-masing dengan datanya sendiri dan kemampuan untuk melakukan perilaku yang sama (seperti menampilkan info), tetapi dengan hasil yang berbeda karena datanya unik.

4) Anda diminta membuat Class BankAccount. Jelaskan bagaimana anda akan menerapkan encapsulation agar data balance tidak bisa diubah sembarangan. Mengapa encapsulation penting untuk keamanan sistem ?

Jawab :

3. menerapkan enkapsulation pada Class BankAccount

Enkapsulasi pada Class BankAccount akan diterapkan untuk melindungi data sensitif seperti balance (saldo) agar tidak bisa diubah secara sembarangan, serta memastikan manipulasi data dilakukan melalui cara yang terkontrol.

Berikut langkah-langkahnya :

* Deklarasi atribut balance sebagai Private :
  - Ini adalah langkah terpenting dalam enkapsulasi. Dengan mendeklarasikan atribut balance sebagai Private kita menyembunyikan detail implementasi internal atribut tersebut dari dunia luar (kelas lain).
  - Artinya, tidak ada code dari luar Class BankAccount yang bisa secara langsung membaca atau mengubah nilai balance.

2. menyediakan metode Publik (Public methods atau Accessors / Mutators) untuk interaksi yang terkont
  - karena balance bersifat Private, kita perlu menyediakan "gerbang" yang aman dan terkontrol bagi kelas lain untuk berinteraksi dengannya. Gerbang ini berbentuk metode Publik.
  - Metode Getter (Accessor) : kita akan menyediakan metode publik seperti getBalance() yang hanya berfungsi untuk mengembalikan nilai balance. Ini memungkinkan kelas lain untuk membaca saldo, tetapi tidak bisa mengubahnya.
  - Metode Mutator (setter) yang terkontrol : Untuk mengubah balance (misalnya, saat deposit atau withdraw), kita tidak akan membuat metode setBalance() umum yang bisa mengubah saldo ke nilai apa pun.
  - misalnya, deposit (double amount) : metode ini akan menambahkan amount ke balance setelah memeriksa apakah amount tersebut positif.
  - misalnya, withdraw (double amount) : metode ini akan mengurangi amount dari balance setelah memeriksa apakah amount positif dan saldo mencukupi.

b. Mengapa enkapsulasi penting untuk keamanan sistem ?

Enkapsulasi adalah pilar fundamental dalam keamanan sistem perangkat lunak, terutama dalam domain finansial seperti perbankan, karena beberapa alasan kritis :

1.) Integritas Data (Data Integrity) :
- mencegah modifikasi yang tidak sah / tidak sengaja
- memastikan konsistensi

2.) Validasi Terpusat (Centralized Validation) :

3.) Pengendalian Akses (Access control)

4.) Mengurangi kompleksitas dan risiko bug :

5.) Memungkinkan perubahan implementasi internal tanpa merusak sistem

5.) Jelaskan bagaimana mekanisme constructor Chaining bekerja pada pewarisan dijava. Apa yang terjadi jika Constructor Pada SuperClass tidak dipanggil secara eksplisit? sertakan ilustrasi Class karyawan dan Subclass Manager.

Jawab :

a. Mekanisme constructor Chaining pada pewarisan di Java

Bayangkan sebuah hubungan antara ayah dan anak. Ketika seorang anak (object subclass) lahir, ia akan mewarisi banyak sifat dari ayahnya (superClass). Namun, agar anak itu, "lengkap" dan bisa berfungsi, bagian dari dirinya yang berasal dari ayah harus diurus atau diinisialisasi terlebih dahulu.

Constructor Chaining adalah proses dimana ketika kita "membangun" sebuah objek anak, proses pembangunan itu selalu dimulai dengan "membangun" bagian dari dirinya yang merupakan warisan dari ayah. Ini terjadi secara berurutan :

1. Alya memutuskan untuk membuat anak (misalnya, "Manager Ana").
2. Sebelum Ana sepenuhnya "terbentuk" sebagai Manager, sistem ini akan memastikan bahwa bagian "karyawan" dari Ana sudah diinisialisasi. Ini seperti memastikan bahwa karakteristik dasar sebagai manusia (seperti nama, identitas) sudah terbentuk sebelum karakteristik spesifik profesi (seperti bonus, manajer) ditambahkan
3. Proses ini berlanjut sampai kelas ke atas : jika karyawan juga "mewarisi" dari identitas yang lebih dasar, maka bagian dasar itu juga akan diinisialisasi terlebih dahulu.
4. Baru setelah semua bagian "ayah" dan "kakek" (superClass, super - superClass, dst.) selesai diinisialisasi, sisa bagian unik dari "anak" (subclass) akan diinisialisasi.

Ini memastikan bahwa setiap bagian dari objek, mulai dari yang paling umum hingga yang paling spesifik, diatur dengan benar dari atas ke bawah (dari superClass ke subClass).

b. Apa yang terjadi jika constructor pada superClass tidak dipanggil secara eksplisit ?

Jika tidak secara eksplisit memanggil konstruktor superClass menggunakan super() di baris pertama konstruktor subClass, compiler Java akan secara otomatis menambahkan panggilan implisit ke konstruktor tanpa argumen (no-argument constructor) dari superclass.

C. Ilustrasi dengan Class karyawan dan subClass manager

- Class karyawan (superClass) :

Bayangkan karyawan sebagai "cetak biru dasar" untuk semua karyawan. Ia mendefinisikan bahwa setiap karyawan punya Nama, ID karyawan, dan Gaji Dasar. Ia juga punya cara untuk "dibuat" (konstruktornya) yang butuh informasi ini. Misalnya, untuk membuat karyawan, Anda harus bilang : "Buat karyawan dengan nama 'Andi', ID 'K001', dan gaji '5 juta'."

- Class Manager (subClass) :

Manager adalah "cetak biru khusus" untuk seorang manajer. Ia adalah jenis karyawan (mewarisi dari karyawan), tapi juga punya karakteristik tambahan, misalnya Bonus.

* Mekanisme Constructor Chaining terjadi saat membuat Manager :

Ketika kita ingin "menciptakan" seorang Manajer bernama Ade :

1. Kita bilang : "Buat Manajer Ade, ID 'M001', gaji dasar '8 juta', bonus '2 juta'."
2. Program tidak langsung membuat Manajer Ade seutuhnya. Ia berpikir : "Manajer itu akan karyawan. Jadi, pertama-tama saya harus membangun bagian karyawan dari Ade dulu."
3. Secara internal, program akan "melompat" ke cetak biru karyawan dan menggunakan instruksi disana untuk membangun bagian dari karyawan dari Ade (yaitu, menetapkan Nama, ID karyawan, dan Gaji Dasar Ade). Ini terjadi karena konstruktor Manager secara eksplisit (atau implisit jika ada) memanggil konstruktor karyawan terlebih dahulu.
4. Setelah bagian "karyawan" dari Ade selesai dibangun, program kembali ke cetak biru Manager. Barulah ia akan melanjutkan dengan menganalisasi atribut spesifik manager, yaitu Bonus sebesar 2 juta.
5. Setelah semua langkah ini, barulah Objek Manggar Ade sepenuhnya terbentuk dan siap digunakan, dengan semua ini karakteristik karyawan dan karakteristik manajer sudah terisi.

Proses "lompat" dari konstruktor subClass ke superClass, lalu kembali lagi, inilah yang disebut Constructor Chaining. Ini memastikan bahwa struktur dan data dari bagian yang diwarisi selalu diinisialisasi dengan benar sebelum bagian spesifik subClass di buat.

6.) Polymorphism memungkinkan kita menulis kode yang fleksibel dan mudah di-maintain. Jelaskan bagaimana penggunaan interface mendukung konsep ini, dan berikan contoh penggunaannya dalam sistem pemesanan online.

Jawab :

a. Polimorfisme dan Peran Interface

Polimorfisme berarti "banyak bentuk". Dalam pemograman, ini adalah kemampuan bagi hal-hal yang berbeda untuk bisa diperlakukan dengan cara yang sama. Bayangkan jika punya sebuah tombol "mainkan" pada perangkat elektronik. Tombol itu sama, tapi ia bisa "memainkan" video di TV, "memainkan" musik di speaker, atau "memainkan" game di konsol. Tombol itu sendiri tidak peduli apa yang dimainkan, selama perangkat yang terhubung tahu cara "memainkan".

Interface (antar muka) itu seperti "kontrak" atau "perjanjian" perilaku. Interface tidak mendefinisikan bagaimana sesuatu bekerja, melainkan hanya apa saja yang harus bisa dilakukan oleh sesuatu jika ia setuju dengan kontrak itu. Misalnya, interface bisa bilang: "Siapa pun yang ingin disebut 'BisaDimainkan' harus punya kemampuan 'Mainkan' dan 'Berhenti'." tapi interface tidak akan menjelaskan bagaimana cara 'maicem' sebuah video atau 'mainkan' sebuah lagu.

b. Bagaimana interface mendukung polimerfisme:

Interface sangat membantu polimerfisme karena ia menciptakan sebuah kesamaan perilaku diantara berbagai "bentuk" yang berbeda.

1. mendefinisikan perilaku umum
2. berinteraksi tanpa detail
3. memungkinkan penggantian

C. Contoh penggunaan dalam sistem pemesanan makanan online

Bayangkan kita sedang membangun sistem pemesanan makanan online. Pelanggan bisa membayar dengan berbagai cara: kartu kredit, dompet digital, atau bahkan transfer bank.

1. membuat "kontrak" (interface):

Kita akan membuat sebuah interface bernama MetodePembayaran. Kontrak ini menyatakan: "Setiap pembayaran yang digunakan dalam sistem ini harus memiliki kemampuan untuk 'memproses pembayaran' dan 'mendapatkan status transaksi'."

2. membuat "bentuk-bentuk" konkret (kelas yang mengimplementasikan interface):

Selanjutnya, kita akan membuat berbagai jenis metode pembayaran yang nyata, dan setiap jenis akan "menyetujui" kontrak MetodePembayaran:

- PembayaranKartuKredit: ini adalah salah satu "bentuk" MetodePembayaran. Ia akan memiliki cara uniknya sendiri untuk "memproses pembayaran" (misalnya, mengirim data ke bank penerbit kartu) dan "mendapatkan status transaksi" (misalnya, menunggu balasan dari bank).
- PembayaranDompetDigital: ini adalah "bentuk" lain. Cara ia "memproses pembayaran" akan berbeda (mungkin memakai API penyedia dompet digital) dan "mendapatkan status transaksi" juga berbeda.
- PembayaranTransferBank: ini adalah "bentuk" ketiga. Cara ia "memproses pembayaran" (mungkin hanya menghasilkan nomor rekening dan instruksi) dan "mendapatkan status transaksi" (mungkin butuh konfirmasi manual atau otomatis) juga akan berbeda.

3. menggunakan "kontrak" dalam sistem utama:

Sekarang, di bagian inti sistem pemesanan makanan yang menangani proses checkout, kita bisa menulis kode yang sangat fleksibel:

- Ketika pelanggan memilih metode pembayaran, sistem akan menerima objek PembayaranKartuKredit, PembayaranDompetDigital, atau PembayaranTransferBank namun, sistem hanya melihatnya sebagai sebuah MetodePembayaran.
- Kode di checkout hanya perlu bilang: "Oke, saya punya sebuah MetodePembayaran. Sekarang, saya akan memintanya untuk 'memproses pembayaran' sebesar x rupiah".
- Sistem tidak perlu tahu atau peduli apakah itu kartu kredit atau dompet digital. Karena semua objek tersebut mengikuti kontrak MetodePembayaran, sistem yakin bahwa fungsi "memproses pembayaran" pasti ada dan bisa dipanggil

Dengan demikian, interface adalah jembatan polimorfisme yang memungkinkan kita menulis code yang umum dan standar, yang dapat berinteraksi dengan berbagai implementasi yang berbeda tanpa harus mengetahui detailnya. Ini membuat sistem sangat adaptif terhadap perubahan dan penambahan baru.

7.) Abstraction membantu menyembunyikan kompleksitas internal. Bandingkan penggunaan abstract class, interface, dan sealed class di Java. Dalam kasus apa masing-masing lebih tepat digunakan?

Jawab:

* Abstraksi dan Perannya dalam menyembunyikan kompleksitas

Abstraksi adalah prinsip desain yang berfokus pada penyajian hanya detail yang esensial kepada pengguna, sambil menyembunyikan implementasi internal yang kompleks. Ibaratnya, ketika kita menggunakan smartphone, kita cukup tahu cara menyentuh layar dan menekan ikon untuk melakukan panggilan atau membuka aplikasi. Kita tidak perlu mengetahui bagaimana sirkuit mikrofon menangkap suara kita, bagaimana data dikonversi menjadi sinyal digital atau bagaimana sinyal tersebut ditransmisikan melalui jaringan seluler. Semua kompleksitas internal itu disembunyikan di balik antarmuka yang sederhana.

Dalam Java, abstract class, interface, dan sealed class adalah alat utama untuk mencapai tingkat abstraksi yang berbeda.

1. Abstract Class (kelas Abstrak)

Abstract class adalah kelas yang tidak dapat diinstansiasi secara langsung (artinya, kita tidak bisa membuat objek dari kelas abstrak). Ia dirancang untuk diwarisi oleh kelas lain (subclass). Abstract class dapat memiliki:

- Atribut (variabel) dan metode konkret (dengan implementasi).
- Metode abstrak (tanpa implementasi, hanya tanda tangan), yang harus diimplementasikan oleh subclass konkretnya.
- konstruktor, meskipun tidak bisa dibuat objeknya

2. Interface (Antarmuka)

Interface adalah kontrak murni. Sebelum Java 8, hanya bisa memiliki:

- metode abstract publik
- metode default
- metode static

3. Sealed Class (kelas Tertutup) - Fitur modern (Java 17+)

Sealed class (atau sealed interface) adalah sebuah kelas atau interface yang memungkinkan kita untuk secara eksplisit membatasi kelas mana yang boleh mewarisi atau mengimplementasikannya. Ini adalah bentuk abstraksi yang lebih terkontrol, dimana kita bisa mendefinisikan hirarki yang terbatas dan diketahui.

Perbandingan dan Ringkasan :

Fitur	Abstract Class	Interface	Sealed Class (atau Interface)
Tipe	kelas	Tipe Referensi (kontrak)	kelas atau interface
Instansiasi	tidak bisa langsung	Tidak bisa langsung	Tidak bisa langsung (jika abstrak / interface)
Atribut	Bisa (interface, static, final)	Hanya konstanta (Public static final)	Bisa (tergantung apakah class / interface)
Metode konkret	Bisa (dengan implementasi)	Bisa (sejak Java 8 : default, static)	Bisa (tergantung apakah class / interface)
Metode abstrak	Bisa (harus diimplementasi subclass)	Bisa (harus diimplementasi kelas implementor)	Bisa (jika abstract class atau interface)
konstruktor	Bisa (dipanggil oleh subclass)	Tidak bisa	Bisa (jika class)
Pewarisan	Satu kelas (extends)	Banyak (implements)	satu (extends) atau banyak (implements)
Tujuan Abstraksi	menyediakan kerangka parsial dan berbagai code / state	mendefinisi kontrak perilaku murni	membatasi dan mengontrol hirarki / implementasi