**پلتفرم ‌مستقل بودن (Platform Independence)**

* **توضیح**: جاوا به‌گونه‌ای طراحی شده که کد نوشته‌شده روی یک سیستم‌عامل یا سخت‌افزار خاص، بدون نیاز به تغییر، روی سیستم‌های دیگر هم اجرا بشه. این ویژگی با شعار "یک‌بار بنویس، همه‌جا اجرا کن" (Write Once, Run Anywhere - WORA) شناخته می‌شه.
* **چگونه کار می‌کنه؟**:
  + کد جاوا به‌جای اینکه مستقیماً به کد ماشین (Machine Code) برای یک سخت‌افزار خاص کامپایل بشه، به بایت‌کد (Bytecode) تبدیل می‌شه. این بایت‌کد توسط **ماشین مجازی جاوا (JVM)** تفسیر و اجرا می‌شه.
  + JVM برای هر پلتفرم (ویندوز، لینوکس، مک و غیره) نسخه خاص خودش رو داره و بایت‌کد رو به دستورات قابل‌فهم برای اون پلتفرم تبدیل می‌کنه.
  + مثلاً، یه برنامه جاوا که روی ویندوز نوشته شده، بدون تغییر روی لینوکس یا حتی دستگاه‌های موبایل (با JVM مناسب) اجرا می‌شه.
* **چرا مهم است؟**:
  + توسعه‌دهندگان نیازی به نوشتن کد جداگانه برای هر سیستم‌عامل ندارن، که باعث صرفه‌جویی در زمان و هزینه می‌شه.
  + این ویژگی جاوا رو برای برنامه‌های وب و اپلیکیشن‌های سازمانی که باید روی سرورها و کلاینت‌های مختلف اجرا بشن، ایده‌آل کرده.
* **مثال در دنیای واقعی**:
  + اپلیکیشن‌های اندروید (که با جاوا یا کاتلین نوشته می‌شن) روی دستگاه‌های مختلف با سخت‌افزارهای متفاوت اجرا می‌شن، چون از JVM (یا معادلش مثل ART در اندروید) استفاده می‌کنن.
  + برنامه‌های سازمانی مثل Apache Tomcat (یه سرور وب جاوایی) روی سرورهای مختلف با سیستم‌عامل‌های گوناگون کار می‌کنن.
* **چالش‌ها**:
  + گاهی تفاوت‌های جزئی در پیاده‌سازی JVMها روی پلتفرم‌های مختلف ممکنه مشکلات کوچکی ایجاد کنه (مثل تفاوت در رندر گرافیکی).
  + عملکرد ممکنه نسبت به زبان‌هایی مثل C++ که مستقیماً به کد ماشین کامپایل می‌شن، کمی کندتر باشه، چون JVM یه لایه اضافی تفسیر داره.
* کد های جاوا با چه کامپایلری به بایت کد تبدیل میشه؟ و "چون JVM یه لایه اضافی تفسیر داره" یعنی چی؟

#### **کامپایلر جاوا که کد رو به بایت‌کد تبدیل می‌کنه**

کدهای جاوا با استفاده از **کامپایلر جاوا (javac)** به بایت‌کد تبدیل می‌شن. توضیح دقیق‌تر:

* **javac چیست؟**: javac (مخفف Java Compiler) بخشی از **JDK (Java Development Kit)** است که کد منبع جاوا (فایل‌های با پسوند .java) رو می‌گیره و به بایت‌کد (فایل‌های با پسوند .class) تبدیل می‌کنه. این بایت‌کد یه زبان میانی (Intermediate Representation) است که مستقل از پلتفرمه و توسط ماشین مجازی جاوا (JVM) اجرا می‌شه.
* **فرایند کار**:
  1. برنامه‌نویس کد جاوا رو در فایلی مثل MyProgram.java می‌نویسه.
  2. با اجرای دستور javac MyProgram.java در ترمینال یا خط فرمان، کامپایلر javac کد رو بررسی می‌کنه (از نظر سینتکس و خطاها) و اگه درست باشه، فایل MyProgram.class رو تولید می‌کنه.
  3. این فایل .class حاوی بایت‌کد است که برای JVM قابل‌فهمه.

##### **معنای "چون JVM یه لایه اضافی تفسیر داره"**

این عبارت به تفاوت نحوه اجرای کد در جاوا نسبت به زبان‌هایی مثل C++ اشاره داره. بیایم دقیق‌تر بررسی کنیم:

* **لایه اضافی تفسیر**:
  + در زبان‌هایی مثل C++، کد منبع مستقیماً به **کد ماشین** (Machine Code) خاص یه پلتفرم (مثل ویندوز یا لینوکس) کامپایل می‌شه. این کد ماشین مستقیماً توسط پردازنده اجرا می‌شه و نیازی به لایه میانی نداره.
  + اما در جاوا، کد منبع ابتدا به **بایت‌کد** (یه زبان میانی) کامپایل می‌شه. این بایت‌کد نمی‌تونه مستقیماً توسط پردازنده اجرا بشه و نیاز به **ماشین مجازی جاوا (JVM)** داره تا اون رو تفسیر کنه یا به کد ماشین تبدیل کنه. این فرایند "لایه اضافی تفسیر" نامیده می‌شه، چون JVM بین بایت‌کد و سخت‌افزار قرار می‌گیره.
* **چگونه کار می‌کنه؟**:
  + JVM بایت‌کد رو می‌خونه و دو روش برای اجرای اون داره:
    1. **تفسیر (Interpretation)**: JVM هر دستور بایت‌کد رو خط‌به‌خط به کد ماشین تبدیل و اجرا می‌کنه. این روش ساده‌تره، اما کندتر.
    2. **کامپایل به‌موقع (Just-In-Time Compilation - JIT)**: JVM از یه کامپایلر JIT استفاده می‌کنه که بایت‌کد رو به کد ماشین بهینه‌شده برای پلتفرم خاص تبدیل می‌کنه. این روش سریع‌تره و در جاوای مدرن بیشتر استفاده می‌شه.
  + این "لایه اضافی" (JVM) باعث می‌شه جاوا پلتفرم‌مستقل باشه، چون بایت‌کد روی هر سیستمی که JVM داشته باشه اجرا می‌شه.
* **چرا بهش "لایه اضافی" می‌گیم؟**:
  + در مقایسه با C++، که کد مستقیماً به کد ماشین تبدیل می‌شه و توسط پردازنده اجرا می‌شه، جاوا یه مرحله میانی (JVM) داره که باید بایت‌کد رو پردازش کنه. این لایه می‌تونه باعث کاهش سرعت اجرا بشه، به‌ویژه در حالت تفسیر (Interpretation) بدون JIT.
  + اما JIT Compiler در JVMهای مدرن (مثل HotSpot JVM) این شکاف عملکرد رو تا حد زیادی کم کرده و گاهی عملکرد جاوا به C++ نزدیک می‌شه.
* **مزایا و معایب این لایه**:
  + **مزایا**:
    1. **پلتفرم‌مستقل بودن**: بایت‌کد روی هر سیستمی با JVM اجرا می‌شه.
    2. **امنیت**: JVM می‌تونه دسترسی کد به منابع سیستم رو کنترل کنه (مثل Sandbox).
    3. **بهینه‌سازی پویا**: JIT Compiler می‌تونه کد رو در زمان اجرا بهینه کنه، مثلاً با حذف کدهای غیرضروری یا بهینه‌سازی حلقه‌ها.
  + **معایب**:
    1. اجرای اولیه ممکنه کندتر باشه، چون JVM باید بایت‌کد رو تفسیر یا کامپایل کنه.
    2. مصرف حافظه بیشتر به‌خاطر نیاز به JVM.
* **مثال**: فرض کن برنامه HelloWorld.java رو کامپایل کردی و فایل HelloWorld.class تولید شده. وقتی دستور java HelloWorld رو اجرا می‌کنی، JVM بایت‌کد این فایل رو می‌خونه و:
  + یا خط‌به‌خط تفسیرش می‌کنه (کندتر).
  + یا با JIT به کد ماشین تبدیلش می‌کنه (سریع‌تر). این فرایند "لایه اضافی" JVM رو نشون می‌ده، چون بدون JVM، بایت‌کد قابل‌اجرا نیست.

**سادگی (Simplicity)**

* **توضیح**: جاوا طوری طراحی شده که یادگیری و استفاده ازش نسبت به زبان‌های پیچیده‌تر مثل C++ ساده‌تر باشه. این سادگی به توسعه‌دهندگان، به‌ویژه مبتدی‌ها، کمک می‌کنه سریع‌تر مفاهیم برنامه‌نویسی رو یاد بگیرن.
* **چگونه محقق شده؟**:
  + **حذف پیچیدگی‌های C++**: جاوا ویژگی‌های پیچیده C++ مثل اشاره‌گرها (Pointers)، ارث‌بری چندگانه (Multiple Inheritance) و مدیریت دستی حافظه رو حذف کرده.
  + **مدیریت خودکار حافظه**: جاوا از **Garbage Collector** استفاده می‌کنه که به‌صورت خودکار حافظه‌ای که دیگه استفاده نمی‌شه رو آزاد می‌کنه. این کار ریسک خطاهایی مثل Memory Leak رو کم می‌کنه.
  + **سینتکس تمیز**: سینتکس جاوا شبیه C هست، اما خواناتر و کمتر مستعد خطاست. مثلاً، تعریف متغیرها و ساختارهای کنترلی (مثل حلقه‌ها و شرط‌ها) ساده و استاندارد شده.
  + **کتابخانه استاندارد غنی**: جاوا مجموعه بزرگی از APIها و کتابخانه‌ها (مثل Java Standard Library) ارائه می‌ده که کارهای رایج مثل کار با فایل‌ها، شبکه یا رابط کاربری رو ساده می‌کنه.
* **چرا مهم است؟**:
  + یادگیری جاوا برای برنامه‌نویسان جدید راحت‌تره، چون نیازی به مدیریت جزئیات سطح پایین (مثل تخصیص حافظه) ندارن.
  + توسعه و نگه‌داری کد سریع‌تر و کم‌خطاتر می‌شه.
* **چالش‌ها**:
  + سادگی گاهی به قیمت انعطاف‌پذیری تموم می‌شه. مثلاً، عدم وجود اشاره‌گرها ممکنه برای برنامه‌نویسان حرفه‌ای که نیاز به کنترل سطح پایین دارن، محدودیت ایجاد کنه.
  + برای پروژه‌های خیلی بزرگ، ممکنه نیاز به درک عمیق‌تر مفاهیم پیشرفته جاوا باشه.

**شیءگرایی (Object-Oriented)**

* **توضیح**: جاوا یه زبان کاملاً شیءگراست، یعنی برنامه‌نویسی در اون حول مفهوم اشیا (Objects) و کلاس‌ها (Classes) می‌چرخه. این ویژگی باعث می‌شه کد سازمان‌یافته‌تر، قابل‌استفاده مجدد و نگه‌داریش راحت‌تر باشه.
* **چگونه پیاده‌سازی شده؟**:
  + **کلاس‌ها و اشیا**: هر برنامه جاوا از کلاس‌ها تشکیل شده که الگویی برای ایجاد اشیا هستن. اشیا نمونه‌هایی از کلاس‌ها هستن که داده‌ها (متغیرها) و رفتارها (متدها) رو ترکیب می‌کنن.
  + **مفاهیم شیءگرایی**:
    - **کپسوله‌سازی (Encapsulation)**: داده‌ها و متدها داخل کلاس‌ها بسته‌بندی می‌شن و با استفاده از Access Modifierها (مثل private، public) کنترل می‌شن.
    - **وراثت (Inheritance)**: کلاس‌ها می‌تونن از کلاس‌های دیگه ارث ببرن تا کد تکراری کم بشه.
    - **چندریختی (Polymorphism)**: امکان استفاده از یک رابط یا متد به شکل‌های مختلف (مثل Overriding یا Overloading).
    - **انتزاع (Abstraction)**: مخفی کردن جزئیات پیچیده و نمایش فقط عملکردهای ضروری با استفاده از کلاس‌های انتزاعی یا رابط‌ها (Interfaces).
  + همه‌چیز در جاوا (به‌جز انواع داده اولیه مثل int) به‌صورت شیء تعریف می‌شه.
* **چرا مهم است؟**:
  + کد شیءگرا ماژولارتره و راحت‌تر می‌شه گسترشش داد یا تغییرش داد.
  + برای پروژه‌های بزرگ و تیمی، ساختار شیءگرا همکاری و نگه‌داری رو ساده‌تر می‌کنه.
* **چالش‌ها**:
  + طراحی شیءگرای خوب نیاز به تجربه داره. اگه کلاس‌ها بد طراحی بشن، ممکنه کد پیچیده و سخت‌فهم بشه.
  + گاهی برای پروژه‌های کوچک، شیءگرایی ممکنه بیش از حد پیچیده به نظر بیاد.

#### **مفاهیم اصلی شیءگرایی**

شیءگرایی حول مفهوم **اشیاء** (Objects) و **کلاس‌ها** (Classes) می‌چرخه. یه شیء ترکیبی از **داده‌ها** (متغیرها یا ویژگی‌ها) و **رفتارها** (متدها) است که از یه کلاس (الگو یا قالب) ساخته می‌شه. چهار اصل اساسی شیءگرایی عبارتند از:

1. **کپسوله‌سازی (Encapsulation)**
2. **وراثت (Inheritance)**
3. **چندریختی (Polymorphism)**
4. **انتزاع (Abstraction)**

هر کدوم از این اصول رو با جزئیات، مثال‌های پیشرفته و تفاوت‌ها توضیح می‌دم.

#### **1. کپسوله‌سازی (Encapsulation)**

* **تعریف**: کپسوله‌سازی یعنی مخفی کردن جزئیات داخلی یه شیء و ارائه یه رابط (Interface) مشخص برای تعامل با اون. این کار با محدود کردن دسترسی به داده‌ها (با استفاده از Access Modifierها مثل private) و ارائه متدهای عمومی (مثل public getterها و setterها) انجام می‌شه.
* **چرا مهم است؟**:
  + **امنیت داده‌ها**: جلوگیری از دسترسی غیرمجاز یا تغییرات ناخواسته.
  + **نگه‌داری آسان‌تر**: تغییرات داخلی کلاس بدون تأثیر روی کدهای خارجی انجام می‌شه.
  + **ماژولار بودن**: کد سازمان‌یافته‌تر و قابل‌فهم‌تر می‌شه.
* **پیاده‌سازی در جاوا**:
  + متغیرها رو private تعریف می‌کنیم و از متدهای public برای دسترسی یا تغییر اون‌ها استفاده می‌کنیم.
  + جاوا از Access Modifierها (private, protected, public, default) برای کنترل دسترسی پشتیبانی می‌کنه.
* **تفاوت با سایر زبان‌ها**:
  + **C++**: مثل جاوا از Access Modifierها پشتیبانی می‌کنه، اما امکان استفاده از اشاره‌گرها (Pointers) می‌تونه کپسوله‌سازی رو تضعیف کنه، چون دسترسی مستقیم به حافظه ممکنه.
  + **پایتون**: کپسوله‌سازی در پایتون ضعیف‌تره، چون متغیرهای private به‌صورت توافقی (با پیشوند \_ یا \_\_) تعریف می‌شن و واقعاً اجباری نیستن.
* **مثال پیشرفته**: فرض کن یه سیستم مدیریت بانکداری داریم که باید حساب‌های بانکی رو مدیریت کنه. می‌خوایم مطمئن بشیم موجودی حساب (Balance) فقط از طریق متدهای خاص تغییر کنه و مستقیماً قابل‌دسترسی نباشه.
  + **نکات پیشرفته**:
    - متغیر accountNumber با final تعریف شده تا غیرقابل‌تغییر باشه.
    - استفاده از synchronized برای جلوگیری از مشکلات هم‌زمانی (Concurrency) در محیط چندنخی.
    - تاریخچه تراکنش‌ها با Collections.unmodifiableList برگردونده می‌شه تا از تغییرات خارجی جلوگیری بشه.
    - کلاس داخلی Transaction برای مدیریت تراکنش‌ها استفاده شده.
  + **چرا سخت؟**: این مثال شامل مدیریت هم‌زمانی، کپسوله‌سازی قوی، و استفاده از کلاس داخلی برای سازمان‌دهی داده‌هاست.

#### **2. وراثت (Inheritance)**

* **تعریف**: وراثت به یه کلاس (زیرکلاس یا Subclass) اجازه می‌ده ویژگی‌ها و متدهای یه کلاس دیگه (فوق‌کلاس یا Superclass) رو به ارث ببره. این کار باعث استفاده مجدد از کد و ایجاد سلسله‌مراتب کلاس‌ها می‌شه.
* **چرا مهم است؟**:
  + **استفاده مجدد کد**: کدهای مشترک در فوق‌کلاس تعریف می‌شن و زیرکلاس‌ها ازش استفاده می‌کنن.
  + **سلسله‌مراتب منطقی**: مدل‌سازی روابط دنیای واقعی (مثل "خودرو" و "کامیون") راحت‌تر می‌شه.
* **پیاده‌سازی در جاوا**:
  + از کلمه‌کلیدی extends برای ارث‌بری استفاده می‌شه.
  + جاوا فقط از **وراثت تکی** (Single Inheritance) پشتیبانی می‌کنه، یعنی یه کلاس فقط می‌تونه از یه فوق‌کلاس ارث ببره (برخلاف C++ که ارث‌بری چندگانه داره).
  + کلمه‌کلیدی super برای دسترسی به متدها یا سازنده‌های فوق‌کلاس استفاده می‌شه.
* **تفاوت با سایر زبان‌ها**:
  + **C++**: از ارث‌بری چندگانه پشتیبانی می‌کنه، که می‌تونه پیچیدگی‌هایی مثل مشکل الماس (Diamond Problem) ایجاد کنه. جاوا با استفاده از رابط‌ها (Interfaces) این مشکل رو حل کرده.
  + **پایتون**: مثل جاوا از ارث‌بری تکی پشتیبانی نمی‌کنه و ارث‌بری چندگانه داره، اما مدیریتش ساده‌تر از C++ است.
* **مثال پیشرفته**: فرض کن یه سیستم گرافیکی برای رندر اشکال هندسی داریم. می‌خوایم اشکال مختلف (مثل دایره و مستطیل) رو مدل کنیم و مساحت و محیطشون رو محاسبه کنیم.
  + **نکات پیشرفته**:
    - کلاس Shape انتزاعی (Abstract) است و متدهای getArea و getPerimeter رو به‌صورت انتزاعی تعریف کرده.
    - زیرکلاس‌ها (Circle و Rectangle) متدهای انتزاعی رو پیاده‌سازی می‌کنن.
    - استفاده از List<Shape> برای ذخیره اشکال مختلف نشون‌دهنده چندریختی هم هست.
    - متد toString برای نمایش اطلاعات شیء بازنویسی شده.
  + **چرا سخت؟**: این مثال شامل کلاس انتزاعی، وراثت، و استفاده از مجموعه‌ها (Collections) برای مدیریت اشیای مختلفه.

#### 3. **چندریختی (Polymorphism)**

* **تعریف**: چندریختی یعنی توانایی یه شیء برای گرفتن اشکال مختلف. این ویژگی به برنامه‌نویس اجازه می‌ده با اشیای مختلف به‌صورت یکسان رفتار کنه (از طریق رابط‌ها یا فوق‌کلاس‌ها) یا متدهای یکسان رو به شکل‌های مختلف پیاده‌سازی کنه.
* **انواع چندریختی**:
  + **چندریختی در زمان کامپایل (Compile-Time Polymorphism)**: مثل Overloading متدها (تعریف متد با امضاهای مختلف).
  + **چندریختی در زمان اجرا (Run-Time Polymorphism)**: مثل Overriding متدها در زیرکلاس‌ها.
* **چرا مهم است؟**:
  + انعطاف‌پذیری: می‌تونی با اشیای مختلف از یه نوع پایه (مثل فوق‌کلاس یا رابط) کار کنی.
  + گسترش‌پذیری: اضافه کردن کلاس‌های جدید بدون تغییر کد موجود.
* **پیاده‌سازی در جاوا**:
  + از طریق **رابط‌ها (Interfaces)** یا **وراثت**.
  + کلمه‌کلیدی @Override برای بازنویسی متدها.
* **تفاوت با سایر زبان‌ها**:
  + **C++**: از چندریختی از طریق وراثت و اشاره‌گرها/مراجع پشتیبانی می‌کنه، اما نیاز به تعریف متدهای مجازی (Virtual) داره.
  + **پایتون**: چندریختی به‌صورت ضمنی (Duck Typing) انجام می‌شه و نیازی به تعریف صریح رابط‌ها نیست.
* **مثال پیشرفته**: فرض کن یه سیستم پردازش پرداخت داریم که باید پرداخت‌های مختلف (کارت اعتباری، کیف پول دیجیتال، ارز دیجیتال) رو مدیریت کنه.
  + **نکات پیشرفته**:
    - رابط PaymentProcessor برای تعریف رفتار مشترک استفاده شده.
    - کلاس‌های CreditCardProcessor و CryptoProcessor متدهای رابط رو به‌شکل خاص خودشون پیاده‌سازی کردن.
    - کلاس PaymentService با رابط کار می‌کنه، نه پیاده‌سازی‌های خاص، که باعث انعطاف‌پذیری می‌شه.
    - این مثال چندریختی در زمان اجرا رو نشون می‌ده.
  + **چرا سخت؟**: این مثال شامل رابط‌ها، چندریختی پویا، و طراحی سیستم ماژولار برای مدیریت انواع مختلف پرداخته.

#### 4. **انتزاع (Abstraction)**

* **تعریف**: انتزاع یعنی مخفی کردن جزئیات پیاده‌سازی و نمایش فقط عملکردهای ضروری به کاربر. این کار از طریق کلاس‌های انتزاعی (Abstract Classes) یا رابط‌ها (Interfaces) انجام می‌شه.
* **چرا مهم است؟**:
  + **سادگی**: کاربر فقط با رابط سطح بالا کار می‌کنه و نیازی به درک جزئیات داخلی نداره.
  + **انعطاف‌پذیری**: می‌تونی پیاده‌سازی‌های مختلف رو بدون تغییر رابط عوض کنی.
* **پیاده‌سازی در جاوا**:
  + **کلاس‌های انتزاعی**: کلاسی که نمی‌تونی مستقیماً ازش شیء بسازی و ممکنه متدهای انتزاعی (بدون پیاده‌سازی) داشته باشه.
  + **رابط‌ها**: قراردادهایی که فقط متدهای بدون بدنه تعریف می‌کنن (هرچند از جاوا 8 به بعد می‌تونن متدهای default یا static هم داشته باشن).
* **تفاوت با سایر زبان‌ها**:
  + **C++**: از کلاس‌های انتزاعی و متدهای مجازی خالص (=0) برای انتزاع استفاده می‌کنه، اما رابط‌ها به‌صورت صریح وجود ندارن.
  + **پایتون**: انتزاع معمولاً با استفاده از کلاس‌های پایه یا ماژول abc (Abstract Base Classes) انجام می‌شه، اما کمتر صلب و اجباریه.
* **مثال پیشرفته**: فرض کن یه سیستم برای مدیریت پروتکل‌های ارتباطی (مثل HTTP و FTP) داریم.
  + **نکات پیشرفته**:
    - کلاس Protocol انتزاعی است و متدهای مشترک (log) و انتزاعی (connect و sendData) داره.
    - کلاس‌های HttpProtocol و FtpProtocol پیاده‌سازی‌های خاص خودشون رو ارائه می‌کنن.
    - این مثال انتزاع و چندریختی رو با هم ترکیب می‌کنه.
  + **چرا سخت؟**: این مثال شامل طراحی سلسله‌مراتب، استفاده از متدهای مشترک و انتزاعی، و مدل‌سازی یه سیستم واقعی (پروتکل‌های شبکه) است.

#### **تفاوت Abstraction و Inheritance**

برای روشن شدن تفاوت‌ها، یه مقایسه مستقیم در جدول زیر ارائه می‌دم:

| **معیار** | **انتزاع (Abstraction)** | **وراثت (Inheritance)** |
| --- | --- | --- |
| **هدف اصلی** | مخفی کردن جزئیات پیاده‌سازی و ارائه رابط سطح بالا. | استفاده مجدد از کد و ایجاد سلسله‌مراتب منطقی بین کلاس‌ها. |
| **تمرکز** | روی «چی» (What) تمرکز داره: چه کاری انجام می‌شه، نه چطور. | روی «به اشتراک گذاشتن» تمرکز داره: انتقال ویژگی‌ها و رفتارها به زیرکلاس‌ها. |
| **ابزار در جاوا** | رابط‌ها (interface) و کلاس‌های انتزاعی (abstract class). | کلمه‌کلیدی extends برای ارث‌بری از کلاس‌های دیگر یا پیاده‌سازی رابط‌ها. |
| **پیاده‌سازی** | رابط‌ها معمولاً بدون پیاده‌سازی (فقط امضا) هستن؛ کلاس‌های انتزاعی می‌تونن پیاده‌سازی نسبی داشته باشن. | زیرکلاس‌ها می‌تونن متدها و ویژگی‌های فوق‌کلاس رو مستقیماً استفاده یا بازنویسی کنن. |
| **رابطه** | رابطه «می‌تواند انجام دهد» (Can-Do): تعریف رفتارهای ممکن بدون مشخص کردن چگونگی. | رابطه «است-یک» (Is-A): زیرکلاس یه نوع خاص از فوق‌کلاسه. |
| **انعطاف‌پذیری** | انعطاف‌پذیرتره، چون رابط‌ها می‌تونن توسط کلاس‌های غیرمرتبط پیاده‌سازی بشن. | محدودتره، چون وابسته به سلسله‌مراتب کلاس‌هاست و جاوا فقط وراثت تکی رو پشتیبانی می‌کنه. |
| **مثال در دنیای واقعی** | رابط Drawable که می‌گه یه شیء می‌تونه رسم بشه، بدون مشخص کردن چطور. | کلاس Car که از Vehicle ارث می‌بره و ویژگی‌های عمومی وسایل نقلیه رو داره. |

##### **توضیح عمیق‌تر تفاوت‌ها**

1. **تمرکز و هدف**:
   * **انتزاع**: هدفش پنهان کردن پیچیدگی‌های پیاده‌سازی و ارائه یه رابط ساده‌ست. مثلاً، وقتی از یه دیتابیس استفاده می‌کنی، فقط می‌خوای بدونی می‌تونی connect و query کنی، نه اینکه پروتکل‌های داخلی چطور کار می‌کنن.
   * **وراثت**: هدفش به اشتراک گذاشتن کد و ایجاد رابطه منطقی بین کلاس‌هاست. مثلاً، یه Car ویژگی‌های عمومی یه Vehicle (مثل سرعت یا برند) رو به ارث می‌بره.
2. **نحوه پیاده‌سازی**:
   * **انتزاع**: از رابط‌ها یا کلاس‌های انتزاعی استفاده می‌کنه که ممکنه هیچ پیاده‌سازی‌ای نداشته باشن (مثل رابط‌ها) یا پیاده‌سازی نسبی داشته باشن (مثل کلاس‌های انتزاعی). این باعث می‌شه کلاس‌های غیرمرتبط بتونن رفتارهای مشابهی داشته باشن (مثل Circle و Rectangle که هر دو Drawable هستن).
   * **وراثت**: مستقیماً کد و رفتار رو از فوق‌کلاس به زیرکلاس منتقل می‌کنه. زیرکلاس معمولاً یه نسخه خاص‌تر از فوق‌کلاسه (مثل Car که یه نوع Vehicle است).
3. **روابط**:
   * **انتزاع**: رابطه‌ای که تعریف می‌کنه بیشتر درباره **قابلیت‌ها** (Capabilities) است. مثلاً، رابط Runnable می‌گه یه شیء می‌تونه اجرا بشه، بدون توجه به اینکه چه نوع شیئیه.
   * **وراثت**: رابطه‌ای که تعریف می‌کنه درباره **هویت** (Identity) است. مثلاً، Dog یه نوع Animal است و ویژگی‌های عمومی حیوانات (مثل eat) رو به ارث می‌بره.
4. **انعطاف‌پذیری**:
   * **انتزاع**: رابط‌ها اجازه می‌دن کلاس‌های کاملاً متفاوت رفتارهای مشترکی داشته باشن (مثل ArrayList و LinkedList که هر دو List رو پیاده‌سازی می‌کنن). این انعطاف‌پذیری در طراحی‌های پیچیده خیلی مفیده.
   * **وراثت**: به‌خاطر وابستگی به سلسله‌مراتب، کمتر انعطاف‌پذیره. مثلاً، در جاوا نمی‌تونی از چند کلاس به‌طور همزمان ارث ببری (مشکل وراثت چندگانه).
5. **پیچیدگی**:
   * **انتزاع**: می‌تونه پیچیدگی رو کاهش بده، چون کاربر فقط با رابط کار می‌کنه و نیازی به درک پیاده‌سازی نداره.
   * **وراثت**: می‌تونه پیچیدگی رو افزایش بده، اگه سلسله‌مراتب خیلی عمیق یا پیچیده بشه (مثل مشکل الماس در زبان‌هایی مثل C++).

##### **مثال پیشرفته برای مقایسه**

برای روشن شدن تفاوت‌ها، یه مثال پیچیده ارائه می‌دم که هم انتزاع و هم وراثت رو نشون می‌ده و تفاوتشون رو برجسته می‌کنه. این مثال یه سیستم مدیریت وسایل نقلیه با قابلیت‌های مختلف (مثل حرکت و سوخت‌گیری) رو مدل می‌کنه.

###### **تحلیل مثال**

* **انتزاع**:
  + رابط Fuelable فقط رفتار سوخت‌گیری رو تعریف می‌کنه، بدون مشخص کردن چگونگی (مثل نوع سوخت).
  + کلاس انتزاعی Vehicle متد move رو به‌صورت انتزاعی تعریف می‌کنه و جزئیات حرکت رو به زیرکلاس‌ها واگذار می‌کنه.
  + کاربر فقط با رابط‌های سطح بالا (Vehicle یا Fuelable) کار می‌کنه و نیازی به دانستن جزئیات پیاده‌سازی (مثل موتور ماشین یا پدال دوچرخه) نداره.
* **وراثت**:
  + کلاس‌های Car و Bicycle از Vehicle ارث می‌برن و متغیر brand و متد move رو به اشتراک می‌گیرن.
  + Car همچنین رابط Fuelable رو پیاده‌سازی می‌کنه، اما Bicycle این کار رو نمی‌کنه، چون دوچرخه نیازی به سوخت‌گیری نداره.
* **تفاوت‌ها در مثال**:
  + **انتزاع**: رابط Fuelable و کلاس انتزاعی Vehicle فقط یه قرارداد (Contract) تعریف می‌کنن (مثل «می‌تونه حرکت کنه» یا «می‌تونه سوخت‌گیری کنه»). این امکان رو می‌ده که اشیای غیرمرتبط (مثل یه پمپ بنزین که Fuelable رو پیاده‌سازی کنه) هم بتونن از این قرارداد استفاده کنن.
  + **وراثت**: Car و Bicycle رابطه «است-یک» با Vehicle دارن و ویژگی‌های مشترکی مثل brand رو به ارث می‌برن. این رابطه سلسله‌مراتبیه و به اشتراک کد کمک می‌کنه.

#### **تفاوت Abstract و Interface**

برای روشن شدن تفاوت‌ها، یه مقایسه مستقیم در جدول زیر ارائه می‌دم:

| **معیار** | **کلاس انتزاعی (Abstract Class)** | **رابط (Interface)** |
| --- | --- | --- |
| **تعریف** | کلاسی که نمی‌تونه نمونه‌سازی بشه و می‌تونه متدهای انتزاعی و معمولی داشته باشه. | قراردادی که رفتارها رو تعریف می‌کنه و معمولاً بدون پیاده‌سازی (مگر متدهای default). |
| **پیاده‌سازی** | با extends ارث‌بری می‌شه. | با implements پیاده‌سازی می‌شه. |
| **وراثت** | جاوا فقط وراثت تکی رو پشتیبانی می‌کنه (یه کلاس فقط از یه کلاس انتزاعی ارث می‌بره). | یه کلاس می‌تونه چندین رابط رو پیاده‌سازی کنه (وراثت چندگانه). |
| **فیلدها** | می‌تونه فیلدهای نمونه (Instance Variables) و ثابت‌ها داشته باشه. | فقط ثابت‌ها (public static final) مجازن، نه فیلدهای نمونه. |
| **متدها** | می‌تونه متدهای انتزاعی و معمولی (با پیاده‌سازی) داشته باشه. | متدهای انتزاعی، default (با پیاده‌سازی)، و static (از جاوا 8). |
| **سازنده‌ها** | می‌تونه سازنده (Constructor) داشته باشه. | نمی‌تونه سازنده داشته باشه. |
| **دسترسی** | می‌تونه متدهای public، protected، یا private داشته باشه. | متدها به‌صورت پیش‌فرض public هستن (نمی‌تونن protected یا private باشن، مگر در موارد خاص از جاوا 9). |
| **هدف اصلی** | ارائه یه کلاس پایه با کد مشترک برای زیرکلاس‌های مرتبط (رابطه Is-A). | تعریف رفتارهای مشترک برای کلاس‌های غیرمرتبط (رابطه Can-Do). |
| **انعطاف‌پذیری** | کمتر انعطاف‌پذیر، چون به سلسله‌مراتب متصله. | انعطاف‌پذیرتر، چون کلاس‌های غیرمرتبط می‌تونن رابط رو پیاده‌سازی کنن. |
| **مثال در دنیای واقعی** | کلاس انتزاعی Vehicle برای ماشین و دوچرخه با رفتار مشترک (مثل brand). | رابط Drawable برای اشیایی که می‌تونن رسم بشن (مثل دایره یا مستطیل). |

##### **توضیح عمیق‌تر تفاوت‌ها**

1. **هدف و کاربرد**:
   * **کلاس انتزاعی**: برای وقتی استفاده می‌شه که می‌خوای یه کلاس پایه با رفتارهای مشترک (متدهای معمولی) و یه سلسله‌مراتب منطقی (رابطه Is-A) ایجاد کنی. مثلاً، Animal برای Dog و Cat که هر دو ویژگی‌های مشترکی مثل name و متد eat دارن.
   * **رابط**: برای تعریف یه رفتار خاص (رابطه Can-Do) استفاده می‌شه که می‌تونه توسط کلاس‌های غیرمرتبط پیاده‌سازی بشه. مثلاً، Runnable برای هر شیئی که بتونه اجرا بشه، چه Thread باشه چه یه کلاس کاملاً متفاوت.
2. **وراثت و انعطاف‌پذیری**:
   * **کلاس انتزاعی**: چون جاوا وراثت تکی رو پشتیبانی می‌کنه، یه کلاس فقط می‌تونه از یه کلاس انتزاعی ارث ببره. این محدودیت می‌تونه طراحی رو سخت کنه.
   * **رابط**: یه کلاس می‌تونه چندین رابط رو پیاده‌سازی کنه، که باعث می‌شه انعطاف‌پذیری بیشتری داشته باشه. مثلاً، یه کلاس می‌تونه هم Runnable و هم Serializable باشه.
3. **فیلدها و حالت**:
   * **کلاس انتزاعی**: می‌تونه حالت (State) داشته باشه (مثل فیلدهای نمونه) و این حالت به زیرکلاس‌ها منتقل می‌شه. مثلاً، Animal می‌تونه name داشته باشه که Dog به ارث می‌بره.
   * **رابط**: نمی‌تونه حالت داشته باشه، فقط ثابت‌ها (static final) مجازن. این باعث می‌شه رابط‌ها سبک‌تر و مناسب‌تر برای تعریف رفتارهای خالص باشن.
4. **متدها**:
   * **کلاس انتزاعی**: می‌تونه متدهای معمولی با پیاده‌سازی کامل داشته باشه، که برای اشتراک کد بین زیرکلاس‌ها عالیه.
   * **رابط**: تا جاوا 7 فقط متدهای انتزاعی داشت، اما حالا با متدهای default می‌تونه پیاده‌سازی هم ارائه بده. بااین‌حال، این متدها معمولاً برای افزودن رفتارهای اختیاری استفاده می‌شن، نه اشتراک کد پیچیده.
5. **سازنده‌ها**:
   * **کلاس انتزاعی**: می‌تونه سازنده داشته باشه که برای مقداردهی اولیه فیلدها در زیرکلاس‌ها استفاده می‌شه.
   * **رابط**: اصلاً سازنده نداره، چون رابط یه شیء نیست و فقط یه قرارداد تعریف می‌کنه.

##### **مثال پیشرفته برای مقایسه**

برای روشن شدن تفاوت‌ها، یه مثال پیشرفته ارائه می‌دم که هم کلاس انتزاعی و هم رابط رو در یه سیستم مدل کنه. این مثال یه سیستم مدیریت اشیای گرافیکی (مثل اشکال) رو نشون می‌ده.

###### **تحلیل مثال**

* **کلاس انتزاعی (Shape)**:
  + یه کلاس پایه برای اشکال فراهم می‌کنه که ویژگی‌های مشترک (مثل color, x, y) و رفتار مشترک (مثل move) رو تعریف می‌کنه.
  + متد انتزاعی getArea به زیرکلاس‌ها (مثل Circle و Rectangle) واگذار شده.
  + رابطه Is-A رو مدل می‌کنه: Circle و Rectangle هر دو یه نوع Shape هستن.
  + چون شامل حالت (State) و متدهای معمولیه، برای اشتراک کد بین اشکال مناسبه.
* **رابط (Drawable)**:
  + رفتار رسم کردن (draw) رو تعریف می‌کنه که می‌تونه توسط هر شیئی (حتی غیرمرتبط با Shape) پیاده‌سازی بشه.
  + متد default (describe) یه رفتار اختیاری اضافه می‌کنه.
  + رابطه Can-Do رو مدل می‌کنه: هر شیء که Drawable رو پیاده‌سازی کنه، می‌تونه رسم بشه.
  + چون حالت نداره، سبک‌تره و برای تعریف رفتارهای خاص مناسبه.
* **ربط به بحث‌های قبلی**:
  + **انتزاع**: هر دو (Shape و Drawable) برای انتزاع استفاده می‌شن، اما Shape با وراثت و اشتراک کد ترکیب شده، درحالی‌که Drawable فقط رفتار تعریف می‌کنه.
  + **وراثت**: Circle و Rectangle از Shape ارث می‌برن (رابطه Is-A)، اما Drawable نیازی به سلسله‌مراتب نداره و می‌تونه توسط کلاس‌های غیرمرتبط پیاده‌سازی بشه.
  + **Has-A**: اگه بخوایم مثلاً یه Canvas به Shape اضافه کنیم (مثل Canvas که Shape رو رسم کنه)، از ترکیب (Has-A) استفاده می‌کنیم، که در این مثال نیست.

##### **کی از Abstract Class و Interface استفاده کنیم؟**

* **از Abstract Class استفاده کن وقتی**:
  + می‌خوای یه کلاس پایه با کد مشترک (متدهای معمولی) و حالت (فیلدها) برای زیرکلاس‌های مرتبط فراهم کنی.
  + رابطه سلسله‌مراتبی منطقی داری (مثل Animal و Dog).
  + می‌خوای بخشی از رفتار رو پیاده‌سازی کنی و بخشی رو به زیرکلاس‌ها واگذار کنی.
  + مثال: وقتی Vehicle برای Car و Truck ویژگی‌های مشترکی مثل brand و متد move داره.
* **از Interface استفاده کن وقتی**:
  + می‌خوای یه رفتار خاص رو تعریف کنی که کلاس‌های غیرمرتبط بتونن پیاده‌سازی کنن.
  + نیاز به وراثت چندگانه داری (چون یه کلاس می‌تونه چندین رابط پیاده‌سازی کنه).
  + نمی‌خوای حالت (State) یا پیاده‌سازی پیچیده به اشتراک بذاری.
  + مثال: وقتی Runnable برای هر شیئی که بتونه اجرا بشه (مثل Thread یا یه کلاس دیگه) استفاده می‌شه.

###### **نکته مهم: ترکیب Abstract Class و Interface**

* در سیستم‌های پیچیده (مثل مثال بیمارستان در پاسخ‌های قبلی)، می‌تونی از هر دو با هم استفاده کنی:
  + از کلاس انتزاعی برای تعریف یه سلسله‌مراتب با کد مشترک (مثل HospitalStaff برای Doctor).
  + از رابط برای تعریف رفتارهای اضافی که ممکنه توسط کلاس‌های مختلف پیاده‌سازی بشن (مثل AppointmentManager برای مدیریت نوبت‌ها).
* مثلاً، در مثال بیمارستان، Doctor از HospitalStaff (کلاس انتزاعی) ارث برد (رابطه Is-A)، اما می‌تونست رابطی مثل Schedulable رو هم پیاده‌سازی کنه تا رفتار خاصی (مثل رزرو نوبت) داشته باشه.

##### **جمع‌بندی**

* **کلاس انتزاعی (Abstract Class)**:
  + یه کلاس پایه با امکان اشتراک کد و حالت.
  + برای سلسله‌مراتب و رابطه Is-A مناسب.
  + می‌تونه متدهای معمولی، فیلدها، و سازنده‌ها داشته باشه.
  + محدود به وراثت تکی در جاوا.
* **رابط (Interface)**:
  + یه قرارداد برای رفتارها بدون حالت.
  + برای رفتارهای مشترک بین کلاس‌های غیرمرتبط و رابطه Can-Do مناسب.
  + پشتیبانی از وراثت چندگانه و انعطاف‌پذیری بیشتر.
  + سبک‌تر، چون فقط ثابت‌ها و متدهای انتزاعی یا default داره.
* **ربط به بحث قبلی (انتزاع و وراثت)**:
  + هر دو برای **انتزاع** استفاده می‌شن، اما کلاس انتزاعی با **وراثت** ترکیب می‌شه و کد مشترکی ارائه می‌ده، درحالی‌که رابط فقط رفتارها رو تعریف می‌کنه و مستقل از سلسله‌مراتبه.
  + در مثال بیمارستان، HospitalStaff (کلاس انتزاعی) برای سلسله‌مراتب و اشتراک کد استفاده شد، و AppointmentManager (رابط) برای تعریف رفتار رزرو نوبت.
* **مثال کلیدی تفاوت**:
  + اگه بخوای اشکال (Shape) با ویژگی‌های مشترک (مثل color) مدل کنی، از کلاس انتزاعی استفاده می‌کنی.
  + اگه بخوای هر شیئی که بتونه رسم بشه (حتی غیراشکال) رو مدل کنی، از رابط Drawable استفاده می‌کنی.

#### **Is A و Has A چیست؟**

##### **تعریف "Is-A"**

* **چیست؟**: رابطه **"Is-A"** (به معنی «است-یک») یه رابطه سلسله‌مراتبیه که با **وراثت (Inheritance)** در برنامه‌نویسی شیءگرا پیاده‌سازی می‌شه. وقتی یه کلاس از کلاس دیگه ارث می‌بره، می‌گیم که زیرکلاس (Subclass) «یه نوع» از فوق‌کلاس (Superclass) است.
* **ویژگی‌ها**:
  + از کلمه‌کلیدی extends در جاوا برای تعریف این رابطه استفاده می‌شه.
  + زیرکلاس همه ویژگی‌ها (متغیرها) و رفتارها (متدها) عمومی یا محافظت‌شده فوق‌کلاس رو به ارث می‌بره.
  + معمولاً با **چندریختی (Polymorphism)** همراهه، چون زیرکلاس می‌تونه متدهای فوق‌کلاس رو بازنویسی (Override) کنه.
* **هدف**:
  + استفاده مجدد از کد (Code Reuse).
  + ایجاد سلسله‌مراتب منطقی (مثل اینکه «ماشین یه وسیله نقلیه‌ست»).
* **مثال ساده**:

java

Copy

* + **توضیح**: Dog یه نوع Animal است (رابطه Is-A). Dog متد makeSound و متغیر name رو از Animal به ارث می‌بره و می‌تونه رفتار خودش رو (مثل پارس کردن) تعریف کنه.

##### **تعریف "Has-A"**

* **چیست؟**: رابطه **"Has-A"** (به معنی «دارد-یک») یه رابطه ترکیبیه که وقتی یه کلاس شامل یه نمونه (Instance) از کلاس دیگه به‌عنوان یه فیلد (Field) باشه، تعریف می‌شه. این رابطه با **ترکیب (Composition)** یا **تجمیع (Aggregation)** پیاده‌سازی می‌شه.
* **ویژگی‌ها**:
  + یه کلاس یه شیء از کلاس دیگه رو به‌عنوان بخشی از خودش نگه می‌داره.
  + ترکیب قوی‌تره (مثل وقتی شیء داخلی با نابودی شیء خارجی نابود می‌شه)، درحالی‌که تجمیع ضعیف‌تره (مثل وقتی شیء داخلی مستقل باقی می‌مونه).
  + نیازی به سلسله‌مراتب نیست؛ فقط رابطه مالکیت (Ownership) مهمه.
* **هدف**:
  + مدل‌سازی روابطی که یه شیء «شامل» یا «مالک» شیء دیگه‌ست (مثل اینکه «ماشین یه موتور داره»).
  + انعطاف‌پذیری بیشتر در طراحی، چون وابستگی به سلسله‌مراتب رو کاهش می‌ده.
* **مثال ساده**:
  + **توضیح**: Car یه Engine داره (رابطه Has-A). Car از Engine ارث نمی‌بره، بلکه یه نمونه ازش رو به‌عنوان فیلد نگه می‌داره و از متدهای اون استفاده می‌کنه.

##### **تفاوت‌های کلیدی بین "Is-A" و "Has-A"**

برای روشن شدن تفاوت‌ها، یه مقایسه مستقیم در جدول زیر ارائه می‌دم:

| **معیار** | **Is-A (وراثت)** | **Has-A (ترکیب/تجمیع)** |
| --- | --- | --- |
| **نوع رابطه** | رابطه سلسله‌مراتبی: زیرکلاس «یه نوع» از فوق‌کلاسه (مثل «سگ یه حیوانه»). | رابطه مالکیت: یه کلاس «شامل» یا «مالک» یه نمونه از کلاس دیگه‌ست (مثل «ماشین یه موتور داره»). |
| **پیاده‌سازی در جاوا** | با کلمه‌کلیدی extends برای ارث‌بری از کلاس یا پیاده‌سازی رابط (implements). | با تعریف یه فیلد (Field) از نوع کلاس دیگه در کلاس اصلی. |
| **هدف اصلی** | استفاده مجدد از کد و ایجاد سلسله‌مراتب منطقی. | مدل‌سازی رابطه مالکیت و افزایش انعطاف‌پذیری در طراحی. |
| **وابستگی** | وابستگی قوی به فوق‌کلاس؛ زیرکلاس به ساختار فوق‌کلاس بستگی داره. | وابستگی کمتر؛ کلاس‌ها مستقل‌ترن و فقط از طریق رابط‌ها یا متدها تعامل می‌کنن. |
| **انعطاف‌پذیری** | کمتر انعطاف‌پذیر، چون به سلسله‌مراتب متصله و جاوا وراثت چندگانه رو پشتیبانی نمی‌کنه. | انعطاف‌پذیرتر، چون می‌تونی اشیاء مختلف رو ترکیب کنی بدون نیاز به سلسله‌مراتب. |
| **مثال در دنیای واقعی** | «سگ یه حیوانه» (Dog is an Animal). | «ماشین یه موتور داره» (Car has an Engine). |
| **پیچیدگی** | می‌تونه با سلسله‌مراتب عمیق پیچیده بشه (مثل مشکل الماس در زبان‌های دیگه). | معمولاً ساده‌تره، چون رابطه مالکیت واضح‌تره و وابستگی‌ها کمترن. |

##### **توضیح عمیق‌تر تفاوت‌ها**

1. **نوع رابطه**:
   * **Is-A**: رابطه‌ایه که نشون می‌ده یه شیء یه نوع خاص از یه دسته بزرگ‌تره. مثلاً، Dog یه Animal است، پس می‌تونه همه ویژگی‌های عمومی حیوانات (مثل خوردن یا حرکت) رو داشته باشه.
   * **Has-A**: رابطه‌ایه که نشون می‌ده یه شیء شامل یا مالک شیء دیگه‌ست. مثلاً، Car یه Engine داره، اما Car خودش یه Engine نیست.
2. **پیاده‌سازی**:
   * **Is-A**: از وراثت استفاده می‌کنه، که یعنی زیرکلاس به‌طور مستقیم به کد و ساختار فوق‌کلاس دسترسی داره. این می‌تونه باعث وابستگی قوی بشه.
   * **Has-A**: از ترکیب استفاده می‌کنه، که یعنی کلاس فقط یه نمونه از کلاس دیگه رو نگه می‌داره و از طریق متدهای عمومی باهاش تعامل می‌کنه. این باعث می‌شه کلاس‌ها مستقل‌تر باشن.
3. **انعطاف‌پذیری**:
   * **Is-A**: به‌خاطر وابستگی به سلسله‌مراتب، انعطاف‌پذیری کمتری داره. مثلاً، در جاوا نمی‌تونی از چند کلاس به‌طور همزمان ارث ببری (مشکل وراثت چندگانه).
   * **Has-A**: انعطاف‌پذیرتره، چون می‌تونی هر تعداد شیء رو به‌عنوان فیلد اضافه کنی و روابط پیچیده رو بدون نیاز به سلسله‌مراتب مدل کنی.
4. **استفاده در طراحی**:
   * **Is-A**: وقتی می‌خوای یه رابطه منطقی و سلسله‌مراتبی بین اشیاء تعریف کنی (مثل حیوانات و زیرمجموعه‌هاشون)، از وراثت استفاده می‌کنی.
   * **Has-A**: وقتی می‌خوای یه شیء شامل اجزای دیگه باشه (مثل یه ماشین که چرخ، موتور، و بدنه داره)، از ترکیب استفاده می‌کنی.
5. **ارتباط با اصول شیءگرایی**:
   * **Is-A**: مستقیماً به **وراثت** و **چندریختی** ربط داره، چون زیرکلاس‌ها می‌تونن رفتارهای متفاوتی برای متدهای ارثی داشته باشن.
   * **Has-A**: بیشتر به **کپسوله‌سازی** ربط داره، چون شیء داخلی (مثل Engine) معمولاً از طریق رابط‌های کنترل‌شده (مثل متد start) مدیریت می‌شه.

##### **مثال پیشرفته برای مقایسه**

برای روشن شدن تفاوت‌ها، یه مثال پیشرفته‌تر ارائه می‌دم که هر دو رابطه **Is-A** و **Has-A** رو در یه سیستم مدل کنه. این مثال یه سیستم مدیریت وسایل نقلیه رو نشون می‌ده که هم وراثت (Is-A) و هم ترکیب (Has-A) توش استفاده شده.

###### **تحلیل مثال**

* **رابطه Is-A**:
  + Car و Bicycle هر دو از Vehicle ارث می‌برن، پس هر دو یه نوع Vehicle هستن (رابطه Is-A).
  + این اجازه می‌ده که متد move در هر کلاس به‌صورت متفاوت پیاده‌سازی بشه (چندریختی).
  + مثلاً، Car یه ماشین خاصه که از ویژگی‌های عمومی Vehicle (مثل brand) استفاده می‌کنه.
* **رابطه Has-A**:
  + Car یه Engine داره (رابطه Has-A)، که به‌عنوان یه فیلد ذخیره شده.
  + Car از متد start کلاس Engine استفاده می‌کنه، اما Engine بخشی از سلسله‌مراتب Vehicle نیست.
  + Bicycle نیازی به Engine نداره، پس این رابطه رو نشون نمی‌ده.
* **ربط به بحث قبلی (انتزاع و وراثت)**:
  + **انتزاع**: کلاس انتزاعی Vehicle یه رابط سطح بالا برای وسایل نقلیه تعریف می‌کنه و جزئیات حرکت (مثل اینکه چطور حرکت می‌کنن) رو مخفی می‌کنه. این به رابطه Is-A کمک می‌کنه، چون زیرکلاس‌ها قرارداد move رو پیاده‌سازی می‌کنن.
  + **وراثت**: رابطه Is-A مستقیماً از وراثت (extends Vehicle) میاد، که کد و رفتار (مثل brand) رو به اشتراک می‌ذاره.
  + **ترکیب (Has-A)**: به وراثت وابسته نیست و به‌جای ارث‌بری، یه شیء (Engine) رو به‌عنوان جزء نگه می‌داره.

##### **کی از Is-A و Has-A استفاده کنیم؟**

* **از Is-A استفاده کن وقتی**:
  + می‌خوای یه رابطه سلسله‌مراتبی و منطقی تعریف کنی (مثل «سگ یه حیوانه»).
  + نیاز به استفاده مجدد از کد و رفتارهای عمومی داری.
  + می‌خوای از چندریختی بهره ببری (مثل رفتارهای مختلف برای متدهای بازنویسی‌شده).
  + مثال: وقتی Car و Truck هر دو نوع Vehicle هستن و رفتارهای مشترکی مثل move دارن.
* **از Has-A استفاده کن وقتی**:
  + می‌خوای یه شیء شامل اجزای دیگه باشه (مثل «ماشین یه موتور داره»).
  + انعطاف‌پذیری بیشتری نیاز داری و نمی‌خوای به سلسله‌مراتب وابسته باشی.
  + وقتی رابطه مالکیت یا ترکیب منطقی‌تر از ارث‌بریه.
  + مثال: وقتی Car شامل Engine، Wheels، و Body است.

###### **نکته مهم: ترجیح ترکیب بر وراثت**

* در طراحی شیءگرا، معمولاً توصیه می‌شه که **ترکیب (Has-A)** رو به وراثت (Is-A) ترجیح بدی، چون:
  + ترکیب انعطاف‌پذیرتره و وابستگی‌ها رو کاهش می‌ده.
  + وراثت می‌تونه باعث وابستگی قوی به فوق‌کلاس بشه و تغییر ساختار سلسله‌مراتب رو سخت کنه.
  + ترکیب بهت اجازه می‌ده رفتارها رو به‌صورت ماژولار اضافه یا حذف کنی (مثل اضافه کردن یه Engine جدید به Car).

##### **جمع‌بندی**

* **Is-A**:
  + رابطه سلسله‌مراتبی که با **وراثت** پیاده‌سازی می‌شه.
  + نشون می‌ده یه کلاس «یه نوع» از کلاس دیگه‌ست (مثل Dog که یه Animal است).
  + وابسته به سلسله‌مراتب و کمتر انعطاف‌پذیر.
  + به **وراثت** و **چندریختی** ربط داره.
* **Has-A**:
  + رابطه مالکیت که با **ترکیب** یا **تجمیع** پیاده‌سازی می‌شه.
  + نشون می‌ده یه کلاس «شامل» یا «مالک» یه شیء دیگه‌ست (مثل Car که یه Engine داره).
  + انعطاف‌پذیرتر و مستقل از سلسله‌مراتب.
  + به **کپسوله‌سازی** ربط داره.
* **ارتباط با بحث قبلی (انتزاع و وراثت)**:
  + **Is-A** مستقیماً به وراثت ربط داره، چون زیرکلاس‌ها از فوق‌کلاس ارث می‌برن و می‌تونن رفتارها رو بازنویسی کنن (مثل مثال Car که از Vehicle ارث می‌بره).
  + **Has-A** به انتزاع نزدیک‌تره، چون می‌تونی از رابط‌ها برای تعریف رفتارهای شیء داخلی استفاده کنی (مثل Engine که می‌تونه رابطی مثل Startable رو پیاده‌سازی کنه).
  + در مثال بیمارستان (پاسخ قبلی)، رابطه Doctor و HospitalStaff یه رابطه Is-A بود (چون Doctor یه نوع HospitalStaff است)، اما Doctor یه لیست Appointment داشت که رابطه Has-A رو نشون می‌داد.

###### **تفاوت‌های چهار اصل**

| **اصل** | **تمرکز اصلی** | **هدف اصلی** | **ابزار در جاوا** | **تأثیر روی طراحی سیستم** |
| --- | --- | --- | --- | --- |
| **کپسوله‌سازی** | مخفی‌سازی داده‌ها و کنترل دسترسی | امنیت و ماژولاریتی داده‌ها | private, getter/setter | کاهش وابستگی و افزایش امنیت |
| **وراثت** | به اشتراک گذاشتن کد و سلسله‌مراتب | استفاده مجدد و ساختار سلسله‌مراتبی | extends, super | سازمان‌دهی کد و گسترش‌پذیری |
| **چندریختی** | تنوع در رفتار با رابط مشترک | انعطاف‌پذیری و تعامل پویا | @Override, interface | امکان تعامل با اشیای مختلف |
| **انتزاع** | مخفی کردن پیاده‌سازی و ارائه رابط | ساده‌سازی و جداسازی رابط از پیاده‌سازی | abstract, interface | ساده‌سازی تعامل و گسترش‌پذیری |

**جمع‌بندی تفاوت‌ها**

* **انتزاع**:
  + روی مخفی کردن جزئیات و ارائه رابط تمرکز داره.
  + با رابط‌ها و کلاس‌های انتزاعی پیاده‌سازی می‌شه.
  + رابطه «می‌تواند انجام دهد» (Can-Do) ایجاد می‌کنه.
  + انعطاف‌پذیرتره و به کلاس‌های غیرمرتبط اجازه می‌ده رفتارهای مشترکی داشته باشن.
* **وراثت**:
  + روی به اشتراک گذاشتن کد و ایجاد سلسله‌مراتب تمرکز داره.
  + با extends پیاده‌سازی می‌شه و کد رو مستقیماً منتقل می‌کنه.
  + رابطه «است-یک» (Is-A) ایجاد می‌کنه.
  + به‌خاطر وابستگی به سلسله‌مراتب، محدودتره.

**مثال های دربرگیرنده همزمان 4 اصل**

**مثال 1: سیستم مدیریت بیمارستان**

این مثال یه سیستم مدیریت بیمارستان رو مدل می‌کنه که شامل پزشکان، بیماران، و نوبت‌دهی است. از هر چهار اصل شیءگرایی استفاده می‌کنه و شامل هم‌زمانی (مرتبط با فصل اول کتاب و بحث‌های قبلی synchronized) هم هست.

**تحلیل**

* **کپسوله‌سازی**:
  + کلاس Patient داده‌های id و appointments رو private کرده و فقط از طریق متدهای addAppointment و getAppointments (که کپی برمی‌گردونه) قابل‌دسترسیه.
  + کلاس Appointment داده‌های patient و doctor رو محافظت می‌کنه.
* **وراثت**:
  + کلاس Doctor از HospitalStaff ارث می‌بره و متد انتزاعی getRole رو پیاده‌سازی می‌کنه.
* **چندریختی**:
  + متد getRole در Doctor بازنویسی شده (Run-Time Polymorphism).
  + کلاس HospitalAppointmentSystem رابط AppointmentManager رو پیاده‌سازی می‌کنه و متدهای خاص خودش رو ارائه می‌ده.
* **انتزاع**:
  + رابط AppointmentManager یه رابط سطح بالا تعریف می‌کنه که جزئیات پیاده‌سازی (مثل نحوه رزرو) رو مخفی می‌کنه.
  + کلاس HospitalStaff یه کلاس انتزاعی برای پرسنل بیمارستانه.
* **چرا پیچیده؟**:
  + ترکیب هم‌زمانی با ReentrantLock برای مدیریت رزروهای همزمان.
  + مدل‌سازی یه سیستم واقعی با روابط پیچیده بین بیمار، پزشک، و نوبت‌ها.
  + استفاده از لیست‌ها و مدیریت حالت‌های مختلف (مثل بررسی در دسترس بودن زمان).

**مثال 2: سیستم پردازش سفارش آنلاین**

این مثال یه سیستم پردازش سفارش برای یه فروشگاه آنلاین رو مدل می‌کنه که شامل محصولات، سبد خرید، و پرداخت است.

**تحلیل**

* **کپسوله‌سازی**:
  + کلاس Product داده‌های id و stock رو private کرده و از AtomicInteger برای مدیریت موجودی به‌صورت اتمیک استفاده می‌کنه.
  + کلاس OrderSystem لیست سبدها رو محافظت می‌کنه.
* **وراثت**:
  + کلاس DiscountedCart از Cart ارث می‌بره و متد calculateTotal رو بازنویسی می‌کنه.
* **چندریختی**:
  + متد calculateTotal در DiscountedCart رفتار متفاوتی نسبت به کلاس پایه داره.
  + رابط PaymentProcessor امکان استفاده از پیاده‌سازی‌های مختلف (مثل CreditCardProcessor) رو فراهم می‌کنه.
* **انتزاع**:
  + رابط PaymentProcessor جزئیات پردازش پرداخت رو مخفی می‌کنه.
  + کلاس Cart یه کلاس انتزاعی برای سبدهای مختلفه.
* **چرا پیچیده؟**:
  + استفاده از ConcurrentHashMap برای مدیریت آیتم‌های سبد خرید در محیط چندنخی.
  + مدیریت موجودی با AtomicInteger برای جلوگیری از Race Condition.
  + ترکیب تخفیف و پردازش پرداخت در یه سیستم واقعی.

**مقایسه دو مثال**

* **مثال 1 (بیمارستان)**:
  + تمرکز روی مدیریت روابط پیچیده بین بیمار، پزشک، و نوبت‌ها.
  + استفاده از ReentrantLock برای هم‌زمانی پیشرفته.
  + مناسب برای سیستم‌هایی با مدیریت منابع محدود (مثل زمان پزشکان).
* **مثال 2 (فروشگاه آنلاین)**:
  + تمرکز روی مدیریت موجودی و پردازش مالی.
  + استفاده از ConcurrentHashMap و AtomicInteger برای عملکرد بالا در محیط چندنخی.
  + مناسب برای سیستم‌های تجارت الکترونیک با تعداد زیاد تراکنش.

**تفاوت‌های کلی جاوا با سایر زبان‌ها در شیءگرایی**

1. **جاوا در برابر C++**:
   * جاوا ارث‌بری چندگانه رو پشتیبانی نمی‌کنه، اما C++ می‌کنه. جاوا به‌جای اون از رابط‌ها استفاده می‌کنه.
   * جاوا اشاره‌گرها رو حذف کرده تا کپسوله‌سازی و امنیت قوی‌تر بشه.
   * جاوا از Garbage Collection برای مدیریت حافظه استفاده می‌کنه، درحالی‌که C++ مدیریت دستی حافظه داره.
2. **جاوا در برابر پایتون**:
   * جاوا شیءگرایی صلب و اجباری داره (همه‌چیز باید داخل کلاس تعریف بشه)، اما پایتون انعطاف‌پذیرتره و از شیءگرایی اختیاری پشتیبانی می‌کنه.
   * جاوا نوع‌بندی قوی (Strong Typing) داره، اما پایتون از نوع‌بندی پویا (Dynamic Typing) استفاده می‌کنه.
3. **جاوا در برابر زبان‌های غیرشیءگرا (مثل C):**
   * جاوا کاملاً شیءگراست و حتی توابع ساده باید داخل کلاس‌ها تعریف بشن، اما C از برنامه‌نویسی رویه‌ای (Procedural) استفاده می‌کنه.

**امنیت (Security)**

* **توضیح**: جاوا از ابتدا با تمرکز روی امنیت طراحی شده تا بتونه برنامه‌های امن برای محیط‌های غیرقابل‌اعتماد (مثل اینترنت) اجرا کنه.
* **چگونه پیاده‌سازی شده؟**:
  + **ماشین مجازی جاوا (JVM)**: JVM یه لایه امنیتی به نام Sandbox فراهم می‌کنه که کدهای جاوا رو در محیطی کنترل‌شده اجرا می‌کنه و دسترسی مستقیم به سیستم‌عامل رو محدود می‌کنه.
  + **مدیر امنیت (Security Manager)**: این ابزار به برنامه‌نویسان اجازه می‌ده سیاست‌های امنیتی خاصی تعریف کنن (مثلاً محدود کردن دسترسی به فایل‌ها یا شبکه).
  + **کنترل نوع (Type Safety)**: جاوا نوع داده‌ها رو به‌شدت بررسی می‌کنه تا از خطاهایی مثل دسترسی غیرمجاز به حافظه جلوگیری کنه.
  + **کتابخانه‌های امنیتی**: جاوا APIهای قوی برای رمزنگاری، احراز هویت و امضای دیجیتال ارائه می‌ده.
  + **حذف اشاره‌گرها**: عدم وجود اشاره‌گرهای مستقیم از خطاهایی مثل Buffer Overflow که در C/C++ رایجه، جلوگیری می‌کنه.
* **چرا مهم است؟**:
  + جاوا برای برنامه‌هایی که روی سرورهای وب یا در محیط‌های حساس (مثل بانکداری آنلاین) اجرا می‌شن، مناسبِ چون ریسک نفوذ و حملات رو کم می‌کنه.
  + این ویژگی جاوا رو برای اپلت‌های قدیمی (که در مرورگرها اجرا می‌شدن) و اپلیکیشن‌های امروزی امن کرده.
* **مثال در دنیای واقعی**:
  + یه اپلیکیشن بانکی که با جاوا نوشته شده، می‌تونه از APIهای امنیتی جاوا (مثل javax.crypto) برای رمزنگاری تراکنش‌ها استفاده کنه تا داده‌ها در برابر هکرها محافظت بشن.
* **چالش‌ها**:
  + تنظیمات امنیتی پیچیده ممکنه برای پروژه‌های ساده وقت‌گیر باشه.
  + گاهی به‌روزرسانی‌های امنیتی برای JVM یا کتابخانه‌ها لازمِ تا آسیب‌پذیری‌های جدید برطرف بشن.

**قابلیت چندنخی (Multithreading)**

* **توضیح**: جاوا از برنامه‌نویسی هم‌زمان (Concurrent Programming) به‌خوبی پشتیبانی می‌کنه، یعنی می‌تونه چندین کار (Thread) رو به‌صورت همزمان اجرا کنه.
* **چگونه پیاده‌سازی شده؟**:
  + **کلاس Thread و رابط Runnable**: جاوا ابزارهای داخلی برای ایجاد و مدیریت نخ‌ها داره.
  + **همگام‌سازی (Synchronization)**: جاوا مکانیزم‌هایی مثل کلمه‌کلیدی synchronized و ابزارهای پیشرفته‌تر مثل java.util.concurrent ارائه می‌ده تا از تداخل نخ‌ها جلوگیری بشه.
  + **مدیریت منابع**: جاوا به برنامه‌نویسان اجازه می‌ده نخ‌ها رو اولویت‌بندی کنن یا منابع سیستم رو بهینه مدیریت کنن.
  + **پشتیبانی از معماری‌های چندپردازنده‌ای**: جاوا از قابلیت‌های چندپردازنده‌ای مدرن بهره می‌بره تا عملکرد بهتری ارائه بده.
* **چرا مهم است؟**:
  + چندنخی برای برنامه‌هایی که نیاز به پردازش موازی دارن (مثل سرورهای وب، بازی‌ها یا اپلیکیشن‌های محاسباتی سنگین) حیاتیه.
  + باعث بهبود عملکرد و پاسخ‌گویی برنامه‌ها می‌شه، به‌ویژه در سیستم‌های چند‌هسته‌ای.
* **چالش‌ها**:
  + مدیریت نخ‌ها می‌تونه پیچیده باشه و خطاهایی مثل Deadlock یا Race Condition ایجاد کنه.
  + نیاز به درک عمیق مفاهیم هم‌زمانی داره تا عملکرد بهینه بشه.

### ****Synchronized چیست؟****

synchronized یه کلمه‌کلیدی در جاواست که برای **کنترل دسترسی همزمان** به منابع مشترک در یه برنامه چندنخی استفاده می‌شه. وقتی چند نخ (Thread) به‌طور همزمان به یه شیء یا متد دسترسی دارن، ممکنه مشکلاتی مثل **Race Condition** (شرایط رقابتی)، **Data Inconsistency** (ناسازگاری داده‌ها)، یا **Corruption** ایجاد بشه. synchronized تضمین می‌کنه که فقط یه نخ در هر لحظه بتونه یه بلوک کد یا متد خاص رو اجرا کنه، و بقیه نخ‌ها باید منتظر بمونن.

#### **چرا synchronized مهم است؟**

در برنامه‌های چندنخی، وقتی چند نخ به یه منبع مشترک (مثل یه متغیر، شیء، یا فایل) دسترسی دارن، ممکنه تغییراتی که یه نخ انجام می‌ده با تغییرات نخ دیگه تداخل کنه. مثلاً، در مثال حساب بانکی که در پاسخ قبلی ارائه شد، اگه دو نخ همزمان بخوایید موجودی حساب رو تغییر بدن (یکی واریز کنه و یکی برداشت)، بدون هماهنگی ممکنه موجودی نادرست بشه. synchronized این مشکل رو با ایجاد **قفل (Lock)** حل می‌کنه.

#### **نحوه عملکرد synchronized**

* **قفل ذاتی (Intrinsic Lock)**: هر شیء در جاوا یه قفل ذاتی (یا Monitor) داره. وقتی یه نخ وارد یه متد یا بلوک synchronized می‌شه، این قفل رو می‌گیره و تا وقتی کارش تموم نشده، قفل رو آزاد نمی‌کنه. بقیه نخ‌ها که می‌خوان به همون منبع دسترسی پیدا کنن، باید منتظر بمونن.
* **دو نوع استفاده از synchronized**:
  1. **متد synchronized**:

public synchronized void myMethod() {

// کد

}

کل متد قفل می‌شه، و قفل روی شیء‌ای که متد بهش تعلق داره (برای متدهای نمونه) یا روی کلاس (برای متدهای static) اعمال می‌شه.

* 1. **بلوک synchronized**:

synchronized (obj) {

// کد

}

فقط کد داخل بلوک قفل می‌شه، و قفل روی شیء مشخص‌شده (obj) اعمال می‌شه. این روش انعطاف‌پذیرتره، چون می‌تونی انتخاب کنی روی کدوم شیء قفل بذاری.

* **نکته کلیدی**: قفل‌ها در جاوا **Reentrant** هستن، یعنی اگه یه نخ قفل یه شیء رو داشته باشه، می‌تونه دوباره همون قفل رو بگیره (مثلاً با فراخوانی یه متد synchronized دیگه) بدون اینکه گیر کنه.

#### **کاربردهای synchronized**

1. **جلوگیری از Race Condition**: وقتی چند نخ به یه متغیر مشترک دسترسی دارن، synchronized تضمین می‌کنه که تغییرات به‌صورت اتمیک (Atomic) انجام بشن.
2. **حفظ سازگاری داده‌ها**: مثلاً، در یه لیست مشترک، اضافه یا حذف آیتم‌ها باید هماهنگ بشه.
3. **مدیریت منابع مشترک**: مثل دسترسی به فایل، پایگاه داده، یا سخت‌افزار.
4. **هماهنگی بین نخ‌ها**: برای اطمینان از اینکه یه نخ منتظر تموم شدن کار نخ دیگه بمونه.

#### **تفاوت synchronized با سایر مکانیزم‌های هم‌زمانی در جاوا**

جاوا ابزارهای پیشرفته‌تری برای مدیریت هم‌زمانی ارائه می‌ده که در بسته java.util.concurrent موجودن. تفاوت‌های synchronized با این ابزارها:

1. **Lock API (مثل ReentrantLock)**:
   * synchronized ساده‌تره، اما انعطاف‌پذیری کمتری داره. مثلاً، نمی‌تونی قفل رو به‌صورت شرطی (Try-Lock) بگیری یا زمان‌بندی کنی.
   * ReentrantLock امکاناتی مثل قفل با زمان انتظار، قفل عادلانه (Fair Lock)، یا باز کردن قفل در متدهای مختلف رو فراهم می‌کنه.

ReentrantLock بخشی از بسته java.util.concurrent.locks است و جایگزین پیشرفته‌تری برای synchronized محسوب می‌شه. این ابزار انعطاف‌پذیری بیشتری برای مدیریت قفل‌ها در برنامه‌های چندنخی ارائه می‌ده.

##### **Lock API (مثل ReentrantLock)**

###### **ویژگی‌ها**

* **Reentrant**: مثل synchronized، اگه یه نخ قفل رو داشته باشه، می‌تونه دوباره همون قفل رو بگیره.
* **انعطاف‌پذیری**:
  + **tryLock()**: می‌تونی قفل رو به‌صورت شرطی بگیری (مثلاً فقط اگه قفل آزاد باشه).
  + **tryLock(long time, TimeUnit unit)**: می‌تونی با زمان انتظار قفل رو بگیری.
  + **Fairness**: می‌تونی قفل رو به‌صورت عادلانه (Fair) تنظیم کنی تا نخ‌ها به‌ترتیب قفل رو بگیرن (کاهش Starvation).
  + **Condition**: امکان ایجاد چندین شرط (مثل wait/notify) برای یه قفل.
* **کنترل دستی**: برخلاف synchronized که قفل به‌صورت خودکار آزاد می‌شه، در ReentrantLock باید قفل رو با unlock() دستی آزاد کنی.
* **تحلیل مثال پیشرفته**:
  + ReentrantLock با fair=true تنظیم شده تا نخ‌ها به‌ترتیب قفل رو بگیرن.
  + Condition برای مدیریت انتظار نخ‌ها استفاده شده (مشابه wait/notify).
  + قفل در بلوک finally آزاد می‌شه تا از گیر کردن قفل جلوگیری بشه.
* **چرا پیشرفته؟**: ترکیب قفل عادلانه، مدیریت شرط‌ها، و شبیه‌سازی یه سناریوی واقعی (صف چاپ).

###### **تفاوت با synchronized**

* **انعطاف‌پذیری**: ReentrantLock قابلیت‌هایی مثل tryLock و Condition داره که در synchronized نیست.
* **کنترل دستی**: نیاز به unlock() صریح داره، که می‌تونه خطرناک باشه اگه فراموش بشه.
* **عملکرد**: در سناریوهای پیچیده، ReentrantLock می‌تونه بهتر عمل کنه، چون قفل‌ها رو دقیق‌تر مدیریت می‌کنه.

##### **Concurrent Collection**

* + به‌جای قفل کردن دستی لیست‌ها با synchronized، می‌تونی از ConcurrentHashMap یا CopyOnWriteArrayList استفاده کنی که برای هم‌زمانی بهینه شدن.

**Concurrent Collections** در بسته java.util.concurrent برای مدیریت مجموعه‌های داده (مثل لیست‌ها، نقشه‌ها، صف‌ها) در محیط‌های چندنخی طراحی شدن. این مجموعه‌ها جایگزین‌هایی برای مجموعه‌های استاندارد (مثل ArrayList یا HashMap) هستن که با synchronized قفل می‌شن.

###### **ویژگی‌ها**

* **بهینه‌سازی برای هم‌زمانی**: به‌جای قفل کردن کل مجموعه، از تکنیک‌هایی مثل تقسیم‌بندی قفل (Lock Striping) یا CAS استفاده می‌کنن.
* **عملکرد بهتر**: برای سناریوهای با تعداد زیاد نخ‌ها، عملکرد بهتری نسبت به قفل‌های synchronized دارن.
* **کلاس‌های کلیدی**:
  + ConcurrentHashMap: نقشه‌ای که برای دسترسی همزمان بهینه شده.
  + CopyOnWriteArrayList: لیستی که برای سناریوهای با خواندن زیاد و نوشتن کم مناسبه.
  + BlockingQueue: صف‌هایی مثل ArrayBlockingQueue یا LinkedBlockingQueue برای تولیدکننده-مصرف‌کننده.
* **تحلیل مثال پیشرفته**:
  + ConcurrentHashMap از Lock Striping استفاده می‌کنه تا فقط بخش‌های خاصی از نقشه قفل بشن، نه کل نقشه.
  + متد compute برای به‌روزرسانی اتمیک استفاده شده.
* **چرا پیشرفته؟**: مدیریت تعداد زیاد دسترسی‌های همزمان با عملکرد بالا.

###### **تفاوت با synchronized Collections**

* Collections.synchronizedMap(new HashMap<>()) کل نقشه رو قفل می‌کنه، که برای تعداد زیاد نخ‌ها کندتره.
* ConcurrentHashMap فقط بخش‌های موردنیاز (Buckets) رو قفل می‌کنه و عملکرد بهتری داره.

##### **Executor Framework**

* + به‌جای مدیریت دستی نخ‌ها و قفل‌ها، می‌تونی از Thread Poolها و ابزارهای java.util.concurrent برای مدیریت وظایف استفاده کنی.

**Executor Framework** در بسته java.util.concurrent یه راه سطح بالا برای مدیریت نخ‌ها و وظایف (Tasks) ارائه می‌ده. به‌جای ایجاد و مدیریت دستی نخ‌ها با Thread، از Thread Poolها و Executorها استفاده می‌شه.

###### **ویژگی‌ها**

* **Thread Pool**: مجموعه‌ای از نخ‌ها که وظایف رو به‌صورت موازی اجرا می‌کنن.
* **انواع Executor**:
  + Executors.newFixedThreadPool(n): یه Thread Pool با تعداد ثابت نخ‌ها.
  + Executors.newCachedThreadPool(): نخ‌ها رو به‌صورت پویا ایجاد و حذف می‌کنه.
  + Executors.newSingleThreadExecutor(): یه نخ برای اجرای وظایف به‌ترتیب.
* **Future و Callable**: برای مدیریت وظایف با خروجی (برخلاف Runnable که خروجی نداره).
* **مزایا**:
  + مدیریت بهینه منابع (مثل CPU و حافظه).
  + کاهش سربار ایجاد نخ‌های جدید.
  + پشتیبانی از وظایف زمان‌بندی‌شده (ScheduledExecutorService).
* **تحلیل مثال پیشرفته**:
  + ExecutorService یه Thread Pool با 4 نخ ایجاد کرده.
  + وظایف با Callable تعریف شدن که خروجی (BigInteger) برمی‌گردونن.
  + Future برای گرفتن نتایج استفاده شده.
* **چرا پیشرفته؟**: مدیریت وظایف سنگین به‌صورت موازی با Thread Pool و گرفتن خروجی‌ها به‌صورت غیرهمزمان.

###### **تفاوت با synchronized**

* synchronized برای هماهنگی دسترسی به منابع مشترکه، اما Executor Framework برای مدیریت اجرای وظایف.
* Executor Framework نیازی به قفل صریح نداره و وظایف رو به نخ‌های موجود در Pool می‌سپاره.

#### **مزیت ها و معایت synchronized**

**مزیت synchronized**

* ساده و مناسب برای سناریوهای معمولی.

**معایب synchronized**

* + عملکرد پایین‌تر در سناریوهای پیچیده (چون قفل کل بلوک یا متد رو نگه می‌داره).
  + عدم انعطاف‌پذیری در مقایسه با Lock API.
  + امکان Deadlock اگه قفل‌ها بد مدیریت بشن.

#### **چالش‌ها و مشکلات synchronized**

1. **Deadlock**: وقتی دو یا چند نخ منتظر قفل‌هایی باشن که همدیگه نگهشون دارن، برنامه گیر می‌کنه.

اگه یه نخ method1 و نخ دیگه method2 رو همزمان اجرا کنه، ممکنه Deadlock پیش بیاد.

1. **Starvation**: یه نخ ممکنه به‌خاطر اولویت پایین‌تر یا رقابت زیاد، هیچ‌وقت قفل رو نگیره.
2. **عملکرد**: قفل کردن کل متد یا شیء ممکنه باعث کاهش عملکرد بشه، چون نخ‌های دیگه باید منتظر بمونن.
3. **Over-Synchronization**: استفاده بیش از حد از synchronized می‌تونه کد رو کند کنه.
4. **تحلیل تست**:
   1. دو نخ Thread-1 و Thread-2 به‌ترتیب method1 و method2 رو اجرا می‌کنن.
   2. method1 اول lock1 و بعد lock2 رو می‌گیره، درحالی‌که method2 اول lock2 و بعد lock1 رو می‌گیره. این ترتیب معکوس باعث Deadlock می‌شه.
   3. یه نخ تشخیص‌دهنده Deadlock اضافه شده که بررسی می‌کنه آیا هر دو نخ در حالت BLOCKED هستن.
   4. خروجی ممکنه نشون بده که نخ‌ها گیر کردن (Deadlock) و برنامه متوقف می‌شه.
5. **چگونه Deadlock رو رفع کنیم؟**:
   1. **ترتیب ثابت قفل‌ها**: همیشه قفل‌ها رو به یه ترتیب بگیر (مثلاً همیشه اول lock1 بعد lock2).
   2. **استفاده از tryLock**: با ReentrantLock می‌تونی از tryLock برای جلوگیری از گیر کردن استفاده کنی.
   3. **زمان‌بندی قفل**: قفل رو با Timeout بگیر تا نخ‌ها بیش از حد منتظر نمونن.

#### **مثال پیشرفته و سطح سخت**

برای توضیح عمیق، یه سیستم مدیریت موجودی انبار (Inventory System) رو مدل می‌کنیم که چند نخ همزمان سعی می‌کنن موجودی یه محصول رو تغییر بدن (مثلاً کم یا زیاد کنن). این سیستم از synchronized برای جلوگیری از ناسازگاری داده‌ها استفاده می‌کنه و شامل مدیریت خطاها و هم‌زمانی پیچیده است.

#### **تحلیل مثال پیشرفته**

* **چرا پیشرفته است؟**:
  + **مدیریت هم‌زمانی**: از synchronized برای قفل کردن دسترسی به inventory استفاده شده تا از Race Condition جلوگیری بشه.
  + **استفاده از wait/notify**: متد removeItem از wait() استفاده می‌کنه تا وقتی موجودی کافی نیست، نخ منتظر بمونه. وقتی موجودی اضافه می‌شه، notifyAll() به همه نخ‌های منتظر اطلاع می‌ده.
  + **AtomicInteger**: برای مدیریت مقدار موجودی به‌صورت اتمیک استفاده شده تا تغییرات کوچک‌تر نیازی به قفل نداشته باشن.
  + **چندنخی واقعی**: سیستم با چند نخ (تأمین‌کننده و مصرف‌کننده) کار می‌کنه که شبیه‌سازی سناریوی واقعی انباره.
* **ربط به مثال قبلی (حساب بانکی)**:
  + در مثال حساب بانکی، synchronized روی متدهای deposit و withdraw استفاده شده بود تا از تغییرات همزمان روی balance جلوگیری کنه. اینجا هم مشابهه، ولی پیچیدگی بیشتره چون شامل انتظار فعال (wait/notify) و مدیریت چند محصوله.
* **چرا سخت؟**:
  + ترکیب synchronized با wait/notify نیاز به درک دقیق هم‌زمانی داره.
  + مدیریت چندین نخ که به یه منبع مشترک دسترسی دارن، پیچیدگی‌های واقعی سیستم‌های توزیع‌شده رو نشون می‌ده.
  + استفاده از AtomicInteger برای بهینه‌سازی و کاهش قفل‌های غیرضروری.

#### **جزئیات عمیق‌تر در مورد synchronized**

1. **انواع قفل‌ها**:
   * **قفل نمونه (Instance Lock)**: وقتی متد synchronized روی یه نمونه (Object) اعمال می‌شه، قفل روی اون شیء خاصه.
   * **قفل کلاس (Class Lock)**: وقتی متد static synchronized باشه، قفل روی کلاس (مثل MyClass.class) اعمال می‌شه.
2. **Reentrancy**: اگه یه نخ قفل یه شیء رو داشته باشه، می‌تونه دوباره همون قفل رو بگیره (مثلاً با فراخوانی یه متد synchronized دیگه).
3. **Visibility**: synchronized علاوه بر قفل کردن، تضمین می‌کنه که تغییرات متغیرها برای همه نخ‌ها قابل‌مشاهده باشه (به‌خاطر Memory Barrier).
4. **محدودیت‌ها**:
   * نمی‌تونی قفل رو به‌صورت دستی آزاد کنی (برخلاف ReentrantLock).
   * نمی‌تونی قفل رو با شرط (مثل Timeout) بگیری.
   * برای سناریوهای پیچیده، ابزارهای java.util.concurrent مثل ReentrantLock یا Semaphore مناسب‌ترن.

#### **مقایسه با سایر زبان‌ها**

* **C++**: از قفل‌های سطح پایین‌تر (مثل Mutex) استفاده می‌کنه. مدیریت دستی‌تره و احتمال خطا بیشتره.
* **پایتون**: از threading.Lock یا threading.RLock برای هم‌زمانی استفاده می‌کنه، اما به‌دلیل GIL (Global Interpreter Lock) در CPython، هم‌زمانی واقعی محدودتره.
* **Go**: از Channelها و Goroutineها برای مدیریت هم‌زمانی استفاده می‌کنه که رویکرد متفاوتی نسبت به قفل‌های صریحه.

### ****مفهوم Atomic در جاوا****

**Atomic** به عملیاتی اشاره داره که به‌صورت **اتمیک (Atomic)** اجرا می‌شه، یعنی یا کامل انجام می‌شه یا اصلاً انجام نمی‌شه، بدون اینکه نخ‌های دیگه بتونن وسطش دخالت کنن. در جاوا، بسته java.util.concurrent.atomic کلاس‌هایی مثل AtomicInteger، AtomicLong، AtomicReference و غیره رو ارائه می‌ده که برای عملیات اتمیک بدون نیاز به قفل‌های صریح (مثل synchronized) طراحی شدن.

#### **ویژگی‌های Atomic**

* **اتمیسیتی (Atomicity)**: عملیات روی متغیرهای اتمیک (مثل افزایش یا کاهش مقدار) به‌صورت کامل و بدون وقفه انجام می‌شه.
* **بدون قفل (Lock-Free)**: برخلاف synchronized که از قفل استفاده می‌کنه، کلاس‌های اتمیک از تکنیک‌های سطح پایین مثل **Compare-And-Swap (CAS)** استفاده می‌کنن که عملکرد بهتری در سناریوهای خاص داره.
* **Visibility**: تغییرات روی متغیرهای اتمیک برای همه نخ‌ها فوراً قابل‌مشاهده است (به‌خاطر Memory Barrier).
* **کاربردها**:
  + شمارشگرها (مثل تعداد کلیک‌ها یا تراکنش‌ها).
  + به‌روزرسانی متغیرهای مشترک بدون نیاز به قفل.
  + جایگزینی برای قفل‌های سنگین در سناریوهای ساده.

#### **چگونه کار می‌کنه؟**

کلاس‌های اتمیک از عملیات CAS (Compare-And-Swap) استفاده می‌کنن که توسط سخت‌افزار پشتیبانی می‌شه. در CAS:

1. مقدار فعلی متغیر با یه مقدار مورد انتظار مقایسه می‌شه.
2. اگه برابر بود، مقدار جدید جایگزین می‌شه.
3. اگه برابر نبود، عملیات دوباره امتحان می‌شه (Optimistic Locking).

#### **مثال پیشرفته**

در مثال سیستم انبار (Inventory System) که قبلاً ارائه شد، از AtomicInteger برای مدیریت موجودی استفاده کردیم. حالا یه مثال پیشرفته‌تر با AtomicReference برای مدیریت یه شیء مشترک می‌نویسم:

* **تحلیل**:
  + AtomicReference برای مدیریت یه شیء Config به‌صورت اتمیک استفاده شده.
  + متد updateAndGet از CAS برای به‌روزرسانی ایمن استفاده می‌کنه.
  + این مثال نشون می‌ده که چطور می‌شه اشیای پیچیده‌تر (نه فقط اعداد) رو به‌صورت اتمیک مدیریت کرد.
* **چرا پیشرفته؟**: استفاده از AtomicReference برای مدیریت اشیای سفارشی و شرطی کردن به‌روزرسانی‌ها (فقط اگه مقدار جدید بزرگ‌تر باشه).

#### **تفاوت با synchronized**

* **عملکرد**: Atomic معمولاً سریع‌تره، چون از CAS استفاده می‌کنه و قفل‌های سنگین رو حذف می‌کنه.
* **انعطاف‌پذیری**: Atomic برای عملیات ساده (مثل افزایش/کاهش) مناسبه، اما برای سناریوهای پیچیده‌تر، synchronized یا Lock بهترن.
* **محدودیت**: کلاس‌های اتمیک فقط برای متغیرهای تک مناسبن و برای هماهنگی چند متغیر نیاز به قفل‌های صریح دارن.

### Stream API چیست؟

Stream یک جریان از اشیاء است که از یک منبع داده (مثل لیست، آرایه، یا Set) تولید می‌شود و امکان انجام عملیات متوالی یا موازی روی داده‌ها را فراهم می‌کند. Stream‌ ها به خودی خود داده‌ها را ذخیره نمی‌کنند، بلکه روی داده‌های منبع عملیات انجام می‌دهند.

##### **ویژگی‌های اصلی:**

1. **غیر مخرب (Non-destructive) : Stream** ها داده‌های اصلی کالکشن را تغییر نمی‌دهند.
2. **پردازش تنبل (Lazy Evaluation)** : عملیات روی Stream تا زمانی که یک عملیات نهایی (Terminal Operation) فراخوانی نشود، اجرا نمی‌شوند.
3. **یک‌بار مصرف:** یک Stream فقط یک‌بار می‌تواند استفاده شود. برای استفاده مجدد، باید Stream جدیدی ایجاد کنید.
4. **پشتیبانی از عملیات موازی:** Stream‌ها می‌توانند به صورت موازی پردازش شوند تا عملکرد در مجموعه‌های بزرگ داده بهبود یابد.

#### **ساختار و عملیات:**

Stream‌ها از سه بخش اصلی تشکیل شده‌اند:

1. **منبع (Source)** : مانند لیست، آرایه، یا هر کالکشن.
2. **عملیات میانی (Intermediate Operations)** : عملیاتی مثل filter، map، sorted که Stream را تغییر می‌دهند و یک Stream جدید برمی‌گردانند.
3. **عملیات نهایی (Terminal Operations)**: عملیاتی مثل collect، forEach، reduce که نتیجه نهایی را تولید می‌کنند و Stream را می‌بندند.

#### **عملیات مهم :**

* **filter(Predicate) :** عناصری که شرط را برآورده می‌کنند، نگه می‌دارد.
* **map(Function) :** هر عنصر را به یک مقدار جدید تبدیل می‌کند.
* **sorted():** عناصر را مرتب می‌کند.
* **collect() :** نتایج را در یک ساختار داده (مثل لیست یا ست) جمع‌آوری می‌کند.
* **forEach(Consumer) :** عملیاتی روی هر عنصر اجرا می‌کند.
* **reduce() :** عناصر را به یک مقدار واحد کاهش می‌دهد (مثل جمع یا ضرب).

#### **۱. مفهوم کلی Stream**

Stream در جاوا یک abstraction است که به شما امکان می‌دهد داده‌ها را به صورت یک جریان (Stream) از اشیاء پردازش کنید، بدون اینکه مستقیماً با ساختار داده اصلی (مثل لیست یا آرایه) کار کنید. Stream API از مفاهیم زیر الهام گرفته شده:

* **Lazy Evaluation (ارزیابی تنبل):** عملیات میانی تا زمانی که یک عملیات نهایی فراخوانی نشود، اجرا نمی‌شوند.
* **Pipeline (خط لوله):** عملیات روی Stream به صورت زنجیره‌ای (Pipeline) اجرا می‌شوند.
* **Immutability:**  Stream‌ها داده‌های منبع را تغییر نمی‌دهند و نسخه‌های جدیدی از داده‌ها تولید می‌کنند.

#### **۲. ساختار داخلی Stream**

Stream API بر پایه چند مؤلفه اصلی ساخته شده است:

* **منبع (Source):** داده‌ها از یک منبع مانند Collection (مثل List یا Set)، آرایه، یا منابع دیگر (مثل Stream.of یا Files.lines) تأمین می‌شوند.
* **Spliterator:** یک رابط داخلی است که منبع داده را به بخش‌های کوچک‌تر تقسیم می‌کند تا پردازش موازی یا متوالی امکان‌پذیر شود. Spliterator مخفف "Splitable Iterator" است و مسئول پیمایش و تقسیم داده‌هاست.
* **Pipeline Operations:** زنجیره‌ای از عملیات میانی و نهایی که روی داده‌ها اعمال می‌شوند.
* **Terminal Sink:** نقطه‌ای که نتایج نهایی جمع‌آوری یا مصرف می‌شوند (مثل collect یا forEach).

#### **۳**. **نحوه عملکرد Stream (گام به گام)**

وقتی از Stream استفاده می‌کنید، مراحل زیر اتفاق می‌افتد:

##### الف) **ایجاد Stream**

Stream از یک منبع داده ایجاد می‌شود. به عنوان مثال:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Stream<Integer> stream = numbers.stream();

* در اینجا، متد stream() از رابط Collection یک نمونه از Stream ایجاد می‌کند.
* در پشت صحنه، جاوا یک **Spliterator** از منبع داده (مثل List) می‌سازد تا داده‌ها را به صورت قابل پیمایش آماده کند.

###### ب) **عملیات میانی (Intermediate Operations)**

عملیات میانی (مانند filter، map، sorted)، Stream را تغییر می‌دهند و یک Stream جدید برمی‌گردانند. این عملیات:

* **تنبل (Lazy) هستند:** تا زمانی که یک عملیات نهایی فراخوانی نشود، هیچ پردازشی انجام نمی‌شود.
* به صورت یک **خط لوله (Pipeline)** ذخیره می‌شوند. مثلاً:

stream.filter(n -> n % 2 == 0).map(n -> n \* 2);

در اینجا، جاوا فقط یک زنجیره از عملیات را ذخیره می‌کند و هیچ داده‌ای پردازش نمی‌شود.

###### ج) **عملیات نهایی (Terminal Operation)**

وقتی یک عملیات نهایی مثل collect یا forEach فراخوانی می‌شود، پردازش واقعی آغاز می‌شود:

* جاوا از **Spliterator**  برای پیمایش داده‌ها استفاده می‌کند.
* عملیات میانی به ترتیب در خط لوله اعمال می‌شوند.
* برای هر عنصر، کل زنجیره عملیات میانی به صورت متوالی یا موازی (در صورت استفاده از parallelStream) اجرا می‌شود.

مثال:

List<Integer> result = numbers.stream()

.filter(n -> n % 2 == 0) // فقط اعداد زوج

.map(n -> n \* 2) // دو برابر کردن

.collect(Collectors.toList()); // جمع‌آوری در لیست

* در اینجا، وقتی collect فراخوانی می‌شود، جاوا هر عنصر را از منبع می‌خواند، آن را از فیلتر (filter) عبور می‌دهد، سپس عملیات نگاشت (map) را اعمال می‌کند و در نهایت نتایج را در یک لیست جمع‌آوری می‌کند.

###### د) **پردازش موازی (Parallel Stream)**

اگر از parallelStream() استفاده کنید:

* **Spliterator** داده‌ها را به بخش‌های کوچک‌تر تقسیم می‌کند.
* هر بخش در یک نخ (Thread) جداگانه پردازش می‌شود (با استفاده از **Fork/Join Framework** در جاوا)
* نتایج بخش‌ها در نهایت ترکیب می‌شوند.

مثال:

List<Integer> result = numbers.parallelStream()

.filter(n -> n % 2 == 0)

.map(n -> n \* 2)

.collect(Collectors.toList());

* اینجا، پردازش روی چندین نخ انجام می‌شود، اما ترتیب خروجی در عملیات مرتب (مثل collect) حفظ می‌شود.

#### **۴**. **اجزای فنی پشت صحنه**

* **ReferencePipeline** : کلاس داخلی که پیاده‌سازی اصلی Stream را ارائه می‌دهد. این کلاس زنجیره عملیات را مدیریت می‌کند.
* **Lambda Expressions :**  Stream API از عبارات لامبدا برای تعریف عملیات (مثل n -> n % 2 == 0 ) استفاده می‌کند.
* **Fork/Join Framework** : برای پردازش موازی، جاوا از این چارچوب استفاده می‌کند تا وظایف را بین نخ‌ها تقسیم کند.
* **Collector :** رابطی که برای جمع‌آوری نتایج (مثل Collectors.toList()) استفاده می‌شود. این رابط شامل متدهایی مثل supplier، accumulator و combiner است که نحوه جمع‌آوری داده‌ها را مشخص می‌کنند.

#### **۵. چگونه جاوا تنبلی (Laziness) را پیاده‌سازی می‌کند؟**

* وقتی عملیاتی مثل filter یا map را فراخوانی می‌کنید، جاوا فقط یک گراف از عملیات (Pipeline) می‌سازد و هیچ داده‌ای را پردازش نمی‌کند.
* این گراف در حافظه به صورت یک زنجیره از اشیاء Stream ذخیره می‌شود.
* وقتی یک عملیات نهایی (مثل collect) فراخوانی می‌شود، جاوا با استفاده از Spliterator داده‌ها را از منبع می‌خواند و کل زنجیره را روی هر عنصر اعمال می‌کند.

#### **۶. مزایا و معایب ساختار Stream**

##### مزایا:

* **بهینه‌سازی خودکار:** به دلیل تنبلی، جاوا می‌تواند عملیات را بهینه کند (مثلاً ترکیبfilter و map در یک پیمایش)
* **پشتیبانی از موازی‌سازی:** بدون نیاز به مدیریت دستی نخ‌ها.
* **کاهش کد تکراری:** کد خواناتر و مختصرتر.
* کد خواناتر و مختصرتر.
* پشتیبانی از برنامه‌نویسی تابع‌محور.
* امکان پردازش موازی با استفاده از parallelStream().

##### **معایب:**

* **پیچیدگی دیباگ:** چون عملیات به صورت زنجیره‌ای و تنبل اجرا می‌شوند، ردیابی خطاها دشوار است.
* **هزینه سربار:** برای داده‌های کوچک، استفاده از Stream ممکن است کندتر از حلقه‌های معمولی باشد.
* **محدودیت یک‌بار مصرف:** Stream پس از یک عملیات نهایی بسته می‌شود.

**محدودیت‌ها:**

* برای داده‌های کوچک، استفاده از Stream ممکن است پیچیدگی غیرضروری ایجاد کند.
* دیباگ کردن Stream ‌ها گاهی دشوار است.

#### **۷. مثال پیشرفته‌تر**

فرض کنید می‌خواهید از یک لیست، اعداد زوج را فیلتر کنید، آن‌ها را دو برابر کنید، مرتب کنید و سپس جمع‌شان را محاسبه کنید:

List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6);

int sum = numbers.stream()

.filter(n -> n % 2 == 0) *// فقط اعداد زوج: [4, 2, 6]*

.map(n -> n \* 2) *// دو برابر: [8, 4, 12]*

.sorted() *// مرتب‌سازی: [4, 8, 12]*

.reduce(0, Integer::sum); *// جمع: 24*

System.out.println(sum); *// خروجی: 24*

* در پشت صحنه، جاوا هر عنصر را یک‌بار از منبع می‌خواند و تمام عملیات (filter، map، sorted، reduce) را به ترتیب روی آن اعمال می‌کند.

#### **۸. جمع‌بندی**

Stream API جاوا با استفاده از Spliterator، Pipeline، و Lazy Evaluation داده‌ها را به صورت تابع‌محور پردازش می‌کند. این API بهینه‌سازی‌های داخلی (مثل ترکیب عملیات و موازی‌سازی) را انجام می‌دهد تا کد خواناتر و کارآمدتر شود. با این حال، برای استفاده بهینه، باید درک خوبی از عملیات میانی و نهایی و محدودیت‌های آن داشته باشید.

مفهوم **"تا وقتی عملیات نهایی انجام نشود، هیچ پردازشی صورت نمی‌گیرد"** به ویژگی **ارزیابی تنبل (Lazy Evaluation)** در Stream API جاوا اشاره دارد. این ویژگی یکی از جنبه‌های کلیدی Stream است که باعث بهینه‌سازی و انعطاف‌پذیری در پردازش داده‌ها می‌شود. اجازه دهید این موضوع را به زبان ساده و با جزئیات توضیح دهم:

#### **مفهوم ارزیابی تنبل (Lazy Evaluation)**

در Stream API، وقتی شما عملیاتی مثل filter، map یا sorted (که عملیات‌های میانی هستند) را روی یک Stream اعمال می‌کنید، این عملیات‌ها **فوراً اجرا نمی‌شوند**. به جای آن، جاوا فقط یک **طرح یا نقشه (Pipeline)** از این عملیات‌ها می‌سازد و منتظر می‌ماند تا یک **عملیات نهایی (Terminal Operation)** مثل collect، forEach یا reduce فراخوانی شود. تنها در این لحظه است که پردازش واقعی داده‌ها آغاز می‌شود.

#### **چرا این اتفاق می‌افتد؟**

این رفتار به دلایل زیر طراحی شده است:

1. **بهینه‌سازی عملکرد:** با به تأخیر انداختن پردازش، جاوا می‌تواند چندین عملیات میانی را در یک پیمایش (traversal) از داده‌ها ترکیب کند، به جای اینکه برای هر عملیات میانی یک پیمایش جداگانه انجام دهد. این کار تعداد دسترسی‌ها به داده‌ها را کاهش می‌دهد و عملکرد را بهبود می‌بخشد.
2. **انعطاف‌پذیری:** ارزیابی تنبل اجازه می‌دهد تا جاوا تصمیم بگیرد که چگونه عملیات‌ها را اجرا کند (مثلاً به صورت متوالی یا موازی) بدون اینکه برنامه‌نویس نیاز به مدیریت این جزئیات داشته باشد.
3. **اجتناب از پردازش غیرضروری:** اگر در نهایت به نتیجه‌ای نیاز نداشته باشید (مثلاً به دلیل یک شرط خاص یا خطا)، جاوا از انجام پردازش‌های غیرضروری جلوگیری می‌کند.

#### **چطور کار می‌کند؟**

وقتی شما یک زنجیره از عملیات میانی را تعریف می‌کنید، Stream فقط این عملیات‌ها را به صورت یک **گراف یا زنجیره** ذخیره می‌کند، بدون اینکه داده‌ها را لمس کند. به محض فراخوانی یک عملیات نهایی، جاوا شروع به خواندن داده‌ها از منبع (مثل لیست یا آرایه) می‌کند و تمام عملیات میانی را به ترتیب روی هر عنصر اعمال می‌کند.

#### **مثال برای روشن شدن موضوع**

**خروجی این کد:**

Stream تعریف شد، اما هنوز پردازش نشده

Filtering: 1

Filtering: 2

Mapping: 2

Filtering: 3

Filtering: 4

Mapping: 4

Filtering: 5

[4, 8]

**توضیح خروجی:**

1. وقتی numbers.stream().filter(...).map(...) اجرا می‌شود، هیچ پیامی چاپ نمی‌شود، چون عملیات‌های filter و map هنوز اجرا نشده‌اند. جاوا فقط یک Pipeline از این عملیات‌ها می‌سازد.
2. وقتی collect(Collectors.toList()) فراخوانی می‌شود، جاوا شروع به پردازش داده‌ها می‌کند:
   * هر عنصر از لیست (1, 2, 3, 4, 5) به ترتیب از منبع خوانده می‌شود.
   * برای هر عنصر، ابتدا filter بررسی می‌کند که آیا عدد زوج است یا نه.
   * اگر عدد زوج باشد، map آن را دو برابر می‌کند.
   * در نهایت، نتایج در یک لیست جمع‌آوری می‌شوند.

##### **نکته کلیدی:**

بدون عملیات نهایی (مثل collect یا forEach)، هیچ داده‌ای پردازش نمی‌شود، چون Stream منتظر دستور نهایی برای شروع کار است. این رفتار باعث می‌شود که:

* پردازش فقط زمانی انجام شود که واقعاً به نتیجه نیاز باشد.
* جاوا بتواند عملیات‌ها را بهینه کند (مثلاً ترکیب filter و map در یک پیمایش).

##### **مثال دیگر: بدون عملیات نهایی**

اگر در مثال بالا خط collect(Collectors.toList()) را حذف کنید:

هیچ پیامی چاپ نمی‌شود، چون هیچ پردازشی انجام نشده است. Stream فقط تعریف شده و منتظر یک عملیات نهایی است.

##### **مزایای این رفتار:**

1. **کاهش تعداد پیمایش‌ها:** به جای پیمایش جداگانه برای filter و map، جاوا هر دو را در یک پیمایش انجام می‌دهد.
2. **پشتیبانی از داده‌های بی‌نهایت:** Stream می‌تواند با منابع داده بی‌نهایت (مثل Stream.generate) کار کند، چون فقط داده‌هایی که نیاز است پردازش می‌شوند.

Stream.generate(() -> 1)

.limit(3) *// فقط 3 عنصر بگیر*

.forEach(System.out::println); *// خروجی: 1, 1, 1*

1. بدون ارزیابی تنبل، پردازش یک جریان بی‌نهایت غیرممکن بود.

### لامبدا (Lambda)

عبارات لامبدا در جاوا (که از نسخه ۸ معرفی شدند) راهی مختصر برای تعریف و استفاده از **توابع بدون نام (Anonymous Functions)** هستند. این عبارات به شما امکان می‌دهند کدهایی که رفتار خاصی را توصیف می‌کنند (مثل یک شرط یا یک تبدیل) به صورت کوتاه و مستقیم بنویسید، بدون نیاز به تعریف کلاس یا متد جداگانه.

#### ساختار یک عبارت لامبدا:

(پارامترها) -> بدنه\_عبارت

* **پارامترها**: ورودی‌هایی که تابع لامبدا می‌گیرد (مثل n یا (x, y)).
* **->**: علامت لامبدا که پارامترها را از بدنه جدا می‌کند.
* **بدنه**: عملیاتی که روی پارامترها انجام می‌شود (مثل n \* 2 یا { return x + y; }).

#### مثال ساده:

(x, y) -> x + y

این لامبدا دو ورودی x و y می‌گیرد و جمع آن‌ها را برمی‌گرداند.

#### **استفاده در Stream:**

لامبدا‌ها در Stream API برای تعریف عملیات‌هایی مثل filter، map، یا forEach استفاده می‌شوند. مثلاً:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

numbers.stream()

.filter(n -> n % 2 == 0) // لامبدا: شرط برای انتخاب اعداد زوج

.map(n -> n \* 2) // لامبدا: دو برابر کردن

.forEach(System.out::println); // خروجی: 4, 8

#### **۲**. **نقش لامبدا در Stream API**

در Stream API، لامبدا‌ها به عنوان پیاده‌سازی **رابط‌های تابعی (Functional Interfaces)** عمل می‌کنند. رابط‌های تابعی، رابط‌هایی هستند که فقط یک متد abstract دارند (مثل Predicate، Function، یا Consumer). لامبدا‌ها به شما امکان می‌دهند بدون تعریف صریح یک کلاس، این متدها را پیاده‌سازی کنید.

##### **رابط‌های تابعی پرکاربرد درStream *:***

1. **Predicate<T>** : برای filter. یک شرط منطقی را بررسی می‌کند (مثل n -> n % 2 == 0).
   * متد: boolean test(T t)
2. **Function<T, R>** : برای map. یک ورودی را به خروجی تبدیل می‌کند (مثل n -> n \* 2).
   * متد: R apply(T t)
3. **Consumer<T>** : برای forEach. عملیاتی روی هر عنصر انجام می‌دهد (مثل System.out::println).
   * متد: void accept(T t)
4. **Supplier<T>:** برای تولید داده (مثل Stream.generate(() -> 1)).
   * متد: T get()

##### **مثال:**

در کد زیر:

numbers.stream().filter(n -> n % 2 == 0)

عبارت n -> n % 2 == 0 یک پیاده‌سازی مختصر از رابط Predicate<Integer> است که متد test را پیاده‌سازی می‌کند:

Predicate<Integer> predicate = new Predicate<Integer>() {

@Override

public boolean test(Integer n) {

return n % 2 == 0;

}

};

لامبدا این کد طولانی را به یک خط کوتاه تبدیل می‌کند.

#### **۳. عملیات پشت صحنه لامبدا**

حالا بیایید به **پشت صحنه** و نحوه کار لامبدا‌ها در جاوا، به‌ویژه در Stream API، بپردازیم. این بخش فنی‌تر است، اما سعی می‌کنم ساده توضیح دهم.

##### **الف) چگونه لامبدا در جاوا پیاده‌سازی می‌شود؟**

لامبدا‌ها در سطح کد منبع (Source Code) به صورت عبارات مختصر نوشته می‌شوند، اما در زمان کامپایل، جاوا آن‌ها را به یکی از دو شکل زیر تبدیل می‌کند:

1. **کلاس‌های داخلی ناشناس(Anonymous Inner Classes) :** در نسخه‌های اولیه جاوا ۸، لامبدا‌ها به کلاس‌های داخلی ناشناس تبدیل می‌شدند. مثلاً:

n -> n \* 2

به چیزی شبیه این تبدیل می‌شد:

new Function<Integer, Integer>() {

@Override

public Integer apply(Integer n) {

return n \* 2;

}

}

اما این روش سربار (Overhead) زیادی ایجاد می‌کرد، چون برای هر لامبدا یک کلاس جدید ساخته می‌شد.

1. **Invokedynamic (از جاوا ۸ به بعد)** : برای بهبود عملکرد، جاوا از دستور **invokedynamic** (معرفی‌شده در جاوا ۷) استفاده می‌کند. این مکانیزم:
   * لامبدا را به یک **متد استاتیک** در زمان کامپایل تبدیل می‌کند.
   * از یک **Lambda Factory** برای ایجاد نمونه‌های سبک‌تر استفاده می‌کند.
   * نتیجه این است که لامبدا‌ها به جای ایجاد کلاس‌های جدید، به متدهای سبک و بهینه تبدیل می‌شوند.

مثلاً، لامبدا n -> n \* 2 ممکن است به یک متد استاتیک در بایت‌کد تبدیل شود:

private static Integer lambda$0(Integer n) {

return n \* 2;

}

سپس، جاوا از invokedynamic برای اتصال این متد به رابط تابعی مناسب (مثل Function) استفاده می‌کند.

##### **ب) چگونه لامبدا در Stream کار می‌کند؟**

وقتی از لامبدا در Stream استفاده می‌کنید، این عبارات به عنوان **قطعات کد قابل اجرا** به Pipeline عملیات Stream پاس داده می‌شوند. بیایید این را با یک مثال بررسی کنیم:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4);

List<Integer> result = numbers.stream()

.filter(n -> n % 2 == 0) // لامبدا 1

.map(n -> n \* 2) // لامبدا 2

.collect(Collectors.toList());

1. **ایجاد Pipeline** :
   * وقتی stream() فراخوانی می‌شود، یک شیء ReferencePipeline (کلاس داخلی جاوا) ایجاد می‌شود.
   * عملیات‌های filter و map به این Pipeline اضافه می‌شوند، همراه با لامبدا‌های مربوطه (n -> n % 2 == 0 و n -> n \* 2).
2. **ذخیره لامبدا‌ها:**
   * هر لامبدا به یک رابط تابعی (مثل Predicate برای filter و Function برای map) متصل می‌شود.
   * این لامبدا‌ها به صورت اشیاء سبک (با استفاده از invokedynamic) در Pipeline ذخیره می‌شوند.
3. **اجرای Pipeline**  :
   * وقتی عملیات نهایی (collect) فراخوانی می‌شود، جاوا با استفاده از **Spliterator** داده‌ها را از منبع (لیست) می‌خواند.
   * برای هر عنصر:
     + لامبدا مربوط به filter (n -> n % 2 == 0) اجرا می‌شود تا بررسی کند آیا عنصر باید نگه داشته شود.
     + اگر عنصر از فیلتر عبور کند، لامبدا مربوط به map (n -> n \* 2) روی آن اعمال می‌شود.
     + نتایج در نهایت توسط collect جمع‌آوری می‌شوند.
4. **بهینه‌سازی:**
   * به دلیل **ارزیابی تنبل**، جاوا تمام عملیات (filter و map) را در یک پیمایش (traversal) انجام می‌دهد.
   * لامبدا‌ها به صورت مستقیم و بدون ایجاد کلاس‌های اضافی (به لطف invokedynamic) اجرا می‌شوند.

##### **ج) موازی‌سازی و لامبدا:**

اگر از parallelStream() استفاده کنید:

* **Spliterator** داده‌ها را به بخش‌های کوچک‌تر تقسیم می‌کند.
* هر بخش در یک نخ جداگانه پردازش می‌شود.
* لامبدا‌ها در هر نخ به طور مستقل روی عناصر اعمال می‌شوند.
* جاوا از **Fork/Join Framework** برای مدیریت نخ‌ها و ترکیب نتایج استفاده می‌کند.

#### **۴. مزایا و معایب لامبدا در Stream**

##### مزایا:

* **مختصر و خوانا:** لامبدا‌ها کد را بسیار کوتاه‌تر و خواناتر می‌کنند.
* **انعطاف‌پذیری:** امکان تعریف رفتارهای مختلف بدون نیاز به کلاس‌های اضافی.
* **بهینه‌سازی داخلی:** استفاده از invokedynamic باعث کاهش سربار و افزایش سرعت می‌شود.

##### معایب:

* **پیچیدگی دیباگ:** چون لامبدا‌ها فشرده هستند، پیدا کردن خطاها ممکن است سخت باشد.
* **محدودیت‌های خوانایی:** در لامبدا‌های پیچیده، کد ممکن است برای مبتدیان گیج‌کننده شود.
* **نیاز به درک رابط‌های تابعی:** برای استفاده مؤثر، باید با رابط‌هایی مثل Predicate وFunction آشنا باشید.

#### **۵**. **مثال پیشرفته‌تر**

فرض کنید می‌خواهید اعداد زوج را فیلتر کنید، آن‌ها را به رشته تبدیل کنید، و سپس با یک جداکننده به هم بچسبانید:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

String result = numbers.stream()

.filter(n -> n % 2 == 0) // لامبدا: Predicate

.map(n -> "Number: " + n) // لامبدا: Function

.collect(Collectors.joining(", ")); // جمع‌آوری به رشته

System.out.println(result); // خروجی: Number: 2, Number: 4

##### پشت صحنه:

* لامبدا n -> n % 2 == 0 به یک متد استاتیک (مثل lambda$filter$0) تبدیل می‌شود و به Predicate.test متصل می‌شود.
* لامبدا n -> "Number: " + n به یک متد استاتیک دیگر (مثل lambda$map$1) تبدیل می‌شود و به Function.apply متصل می‌شود.
* وقتی collect فراخوانی می‌شود، جاوا هر عنصر را از Spliterator می‌خواند، لامبدا‌ها را روی آن اعمال می‌کند، و نتایج را با Collectors.joining ترکیب می‌کند.

#### **۶. جمع‌بندی**

* **لامبدا چیست؟** راهی مختصر برای تعریف توابع بدون نام که به عنوان پیاده‌سازی رابط‌های تابعی (مثل Predicate یا Function) عمل می‌کنند.
* **نقش در Stream؟** لامبدا‌ها رفتار عملیات میانی (مثل filter و map) و نهایی (مثل forEach) را تعریف می‌کنند.
* **پشت صحنه؟** لامبدا‌ها با استفاده از invokedynamic به متدهای استاتیک بهینه تبدیل می‌شوند و در Pipeline عملیات Stream اجرا می‌شوند. Spliterator و Fork/Join Framework برای مدیریت داده‌ها و موازی‌سازی استفاده می‌شوند.