অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং: ইন্টারফেস এবং পলিমর্ফিজম

shafaetsplanet.com/

শাফায়েত

জুন ২, ২০১৯

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং টপিকটা খুব সহজ কিছু না, এই সিরিজে আমি চেষ্টা করবো তোমার জন্য টপিকটা কিছুটা সহজ করে দেয়ার। অ্যালগরিদম আর ডাটা স্ট্রাকচার শেখার পর সম্ভবত সবথেকে গুরুত্বপূর্ণ হলো অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং শেখা। “কাজ করে” এমন কোডতো সবাই লিখতে পারে, কিন্তু তুমি যদি এমন লিখতে চাও যেখানে কিছুদিন পরপর নতুন ফিচার যোগ করতে হয়, যার পিছে অনেকজন একসাথে কাজ করছে তাহলে যেনতেন ভাবে কোড লিখলে চলবে না, তোমাকে সঠিক ডিজাইন করা জানতে হবে, এখানেই অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং জানাটা জরুরী হয়ে যায়। অ্যালগরিদম নিয়ে অনেক বছর লেখালেখি পর মনে হলো এখন এসব নিয়েও লেখা দরকার।

অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং শিখতে গিয়ে প্রথমে বেশ কঠিন লাগে। এটার মূল কারণ বেসিক কনসেপ্টগুলো ভালো ভাবে ক্লিয়ার না হওয়া। তুমি যদি ক্লাস আর অবজেক্টের পার্থক্য না বুঝো, ইন্টারফেস আর অ্যাবস্ট্র্যাক্ট ক্লাসের পার্থক্য না বুঝো তাহলে যখন ওওপি নিয়ে কোনো লেখা পড়তে যাবে সেগুলো দুর্বোধ্য লাগবে এবং যখন ডিজাইন প্যাটার্ন শেখার চেষ্টা করবে তখন আরো কঠিন লাগবে।

এই লেখা পড়ার সময় আমি আশা করবো তুমি অন্তত ক্লাস এবং অবজেক্ট সম্পর্কে জানো। তুমি জানো কিভাবে ক্লাসের ভিতর মেথড বা ভ্যারিয়েবল ডিফাইন করা হয়, কিভাবে ক্লাসের ইন্সট্যান্স তৈরি করা হয়, কনস্ট্রাক্টর কি। এগুলো না জানলে বা ভুলে গেলে একটু ঘাটাঘাটি করে আসো। আমি এই সিরিজে জাভা ব্যবহার করবো।

অনেকে Animal, Food এই ধরণের ক্লাস ব্যবহার করে ওওপি শেখানোর চেষ্টা করে। আমি সেটার পক্ষপাতি না, আমার মতে আরেকটু প্র্যাকটিকাল উদাহরণ ব্যবহার না করলে জিনিসটা মাথায় ঢুকে না। আমি শুধু সংজ্ঞা দিবো না, চেষ্টা করবো তোমাকে আরেকটু গভীরে নিয়ে যাওয়ার।

আজকে আমরা প্রথমে ইন্টারফেস সম্পর্কে জানবো। অনেকেই ইনহেরিটেন্স দিয়ে আগে শুরু করে, তাতে কোনো সমস্যা নাই। তবে আমি যেভাবে ব্যাখ্যা করতে চাই তাতে ইন্টারফেসটা নিয়ে আগে বলা গুরুত্বপূর্ণ। অবজেক্ট অরিয়েন্টেড প্রোগ্রামিং এর ৪টা মুল ভিত্তি আছে যার দুটি হলো **polymorphism** আর **abstraction** আজকের লেখা পড়ে তুমি এগুলো সম্পর্কেও জানতে পারবে। ইন্টারফেস না শিখিয়ে আমি যদি সরাসরি এগুলোর সংজ্ঞায় চলে যেতাম তাহলে তুমি কিছুই বুঝতে পারতে না।

ইন্টারফেস কি?

শুরুতে প্রোগ্রামিং এর কথা ভুলে যাও, আমরা বাস্তব জীবনেও ইন্টারফেস শব্দটা অহরহ ব্যবহার করি। বাস-ট্রাকের কথা চিন্তা করো। যে বাস চালাতে পারে সে কিন্তু ট্রাকও চালাতে পারে। বাস-ট্রাক বা গাড়ির চালানোর সময় কাজ কিন্তু একই, প্রথমে ইঞ্জিন চালু করতে হয়, এরপর এক্সেলেটরে পা দিয়ে সেটাকে সামনে নিতে হয়, স্টিয়ারিং দিয়ে বামে-ডানে নিতে হয়। এখন ভলভো বাসের স্টিয়ারিং আর টয়োটা গাড়ির স্টিয়ারিং কিন্তু পুরোপুরি একই ভাবে কাজ করে না, হুডের ভিতরে মেকানিজম আলাদা। কিন্তু আমরা হুডের ভিতর গিয়ে দেখিনা কে কিভাবে স্টিয়ারিং “ইম্প্লিমেন্ট” করেছে, আমাদেরকে একটা স্টিয়ারিং নামের ইন্টারফেস ধরিয়ে দিয়ে বলা হয়েছে এর দুটি ফিচার আছে “বামে যাও”, “ডানে যাও”, তুমি এগুলো ব্যবহার করে চালাও, হুডের ভিতর কিভাবে কাজ করে সেটা তোমার মাথাব্যাথা না।

তো ইন্টারফেস হলো একটা সিস্টেমের সাথে যোগাযোগ করার একটা লেয়ার। তুমি মোবাইলে বিভিন্ন বাটনে চাপ দিচ্ছো সেটাও একটা ইন্টারফেস, ভিতরে কিভাবে কাজ হচ্ছে তুমি জানো না, তুমি শুধু জানো কোন বাটনের কাজ কি। তো একটা ইন্টারফেসের দুটি অংশ থাকে:

- ইন্টারফেসের ডেফিনেশন। সেখানে বলা হয় ইন্টারফেসটা কি কি কাজ করতে পারে। কাজগুলো তোমার কাছে

"অ্যাবস্ট্র্যাক্ট", তুমি জানো না ভিতরে কি ঘটছে।
- ইন্টারফেসের ইমপ্লিমেন্টেশন, সেখানে বলা হয় ইন্টারফেসটা ঠিক কিভাবে কাজ করে।

প্রোগ্রামিং এর জগতে ইন্টারফেস

ইন্টারফেস হলো এমন একটা ক্লাস যার ভিতর কিছু মেথড বা ভ্যারিয়েবল আছে কিন্তু সেগুলার ডেফিনেশন দেয়া নেই।

যেমন ধরো আমি হয়তো একটা লাইব্রেরি বানাচ্ছি যেটার কাজ হলো কিছু সংখ্যার যোগফল বের করার। আমার লাইব্রেরিতে দুটি মেথড আছে। একটি মেথড ব্যবহার করে তুমি নতুন সংখ্যা যোগ করতে পারবে, আরেকটা মেথড ব্যবহার করে এখন পর্যন্ত পাওয়া সবগুলো সংখ্যাগুলোর যোগফল জানতে পারবে।

আমি নিচের মতো করে একটা ইন্টারফেস ডিফাইন করবো:

```
public interface SummationService {
  void addNew(int num);
  int findSum();
}


```

এটা একটা ইন্টারফেস দিয়ে আমি শুধু ঠিক করে দিলাম যে *SummationService* কি কি ধরণের সার্ভিস দিবে। আমি দুটি মেথডের সিগনেচার ডিক্লেয়ার করেছি কিন্তু মেথডটা কিভাবে কাজ করে সেটা ডিফাইন করিনি। এই ধরনের মেথডকে বলে abstract method। ***Abstraction*** হলো ইমপ্লিমেন্টেশন ডিটেইলস লুকিয়ে খালি ফিচারগুলো ইউজারকে দেয়া। তাই যে method এর কি করে আমরা সেটা জানি কিন্তু কিভাবে করে সেটা জানি না সেটাই হলো Abstract Method।

এখন আমরা আরেকটু কঠিন করে ইন্টারফেসের সংজ্ঞা দিতে পারি, ইন্টারফেস হলো এমন একটা ক্লাস যেটার ভিতর শুধুমাত্র কিছু *abstract method* আছে।

এখন আমার এমন একটা ক্লাস লাগবে যে ইন্টারফেসের মেথডগুলোকে ***implement*** করবে। শুধুমাত্রই তখনই আমরা ইন্টারফেস ব্যবহার করে কিছু একটা করতে পারবো।

জাভা ইন্টারফেস ২
Java

```
public class SimpleSum implements SummationService {
}
```

এখানে ***implements*** শব্দটা লক্ষ্য করো। এটা দিয়ে বুঝানো হয়েছে *SimpleSum* ক্লাসটি *SummationService* এ যেসব মেথড ডিক্লেয়ার করা হয়েছে সেগুলো ইমপ্লিমেন্ট করে দিবে। কিন্তু আমাদের ক্লাসের ভিতর এখন কিছুই নেই তাই তুমি যদি এই ক্লাসটাকে কম্পাইল করার চেষ্টা করো তাহলে একটা এরোর মেসেজ পাবে:

```
Class 'SimpleSum' must either be declared abstract
or implement abstract method 'addNew(int)' in 'SummationService'
```

"either be declared abstract" এই অংশটা আপাতত ভুলে যাও। এরোর মেসেজে বলছে *SimpleSum* কে অবশ্যই *addNew(int)* নামক মেথডকে ইমপ্লিমেন্ট করতে হবে। না করলে "চুক্তিভঙ্গ" হবে। ইন্টারফেসকে অনেকসময় এজন্য বলা হয় contract, তুমি যদি ইন্টারফেস ইমপ্লিমিন্ট করো তাহলে অবশ্যই contract মেনে সবগুলো মেথড

ইম্প্লিমেন্ট করতে হবে।

আমরা তাহলে মেথডগুলোকে ইম্প্লিমেন্ট করে ফেলি:

```
public class SimpleSum implements SummationService {
  private int sum;
  SimpleSum() {
    sum = 0;
  }
  @Override
  public void addNew(final int num) {
    sum = sum + num;
  }
  @Override
  public int findSum() {
    return sum;
  }
}
```

এইবার *SimpleSum* ক্লাসটি সবগুলো মেথড ইম্প্লিমেন্ট করেছে। @Override অ্যানোটেশনটা এখানে ব্যবহার করা ঐচ্ছিক, এটা দিয়ে বুঝাচ্ছে এই মেথডগুলো ইন্টারফেসের ক্লাসকে ইমপ্লিমেন্ট করছে, অ্যানোটেশন নিয়ে বিস্তারিত এখানে আলোচনা করবো না।

এখন আমি আরেক ধরনের ইমপ্লিমেন্টেশন চাই যেটা সবগুলো সংখ্যা যোগ না করে শুধুমাত্র ইউনিক সংখ্যাগুলোর যোগফল বলে দেয়। সেক্ষেত্রে আমাদের অ্যালগরিদম একটু অন্যরকম হবে, আমরা সংখ্যাগুলোকে একটা সেট এ ভরে রাখবো যাতে প্রতিবার নতুন সংখ্যা পেলে আমরা চেক করে দেখতে পারি যে সংখ্যাটি আগে পেয়েছি নাকি। আমরা *UniqueSum* নামের একটা ক্লাস তৈরি করি:

```
public class UniqueSum implements SummationService {
  private Set<Integer> uniqueNumbers;
  private int sum;
  UniqueSum() {
    this.uniqueNumbers = new HashSet<>();
    sum = 0;
  }
  @Override
  public void addNew(final int num) {
    if (!uniqueNumbers.contains(num)) {
      uniqueNumbers.add(num);
      sum += num;
    }
  }
  @Override
  public int findSum() {
    return sum;
  }
}
```

আমরা দুটি ক্লাস পেয়েছি যারা ইন্টারফেসটিকে ইম্প্লিমেন্ট করে। এখন আমরা দেখি কিভাবে ইন্টারফেসটা ব্যবহার করা যায়:

```
public class ExampleUsage {
  public static void main(String[] args) {
    final SummationService simpleSummationService = new SimpleSum();
    final SummationService uniqueSummationService = new UniqueSum();
    simpleSummationService.addNew(10);
    simpleSummationService.addNew(10);
    simpleSummationService.addNew(20);
    System.out.println(simpleSummationService.findSum());
    uniqueSummationService.addNew(10);
    uniqueSummationService.addNew(10);
    uniqueSummationService.addNew(20);
    System.out.println(uniqueSummationService.findSum());
  }
}
```

এখানে দেখো আমি কিভাবে দুইরকমের *SummationService* তৈরি করেছি। একবার আমি ব্যবহার করেছি *SimpleSum* এর কনস্ট্রাক্টর, আরেকবার *UniqueSum* এর। কিন্তু দুইবারই আমি অ্যাসাইন করেছি একই টাইপের ভ্যারিয়েবল এ। তারমানে *SummationService* ইন্টারফেসটা দুইরকম ইমপ্লিমিন্টেশন অনুযায়ি দুইরকম রূপ ধারণ করেছে।

এই জিনিসটাকে বলা হয় **Polymorphism** যার মানে হলো একাধিক রূপ ধারণ করার ক্ষমতা। আমরা এখনই দেখবো এই প্রোপার্টি ব্যবহার করে কিভাবে ক্লাস ডিজাইন করা যায়। আর *polymorphic object* হলো এমন একটা অবজেক্ট যেটা একাধিক রূপ ধারণ করতে পারে।

এবার আমি একটা ক্লাস তৈরি করতে চাই *DataProcessor*। এই ডাটা প্রসেসর বিভিন্ন রকমের ডাটা প্রসেসিং এর কাজ করে এবং কাজ করার জন্য *SummationService* এর সাহায্য নেয়। ক্লাসটা হতে পারে এরকম:

```
class DataProcessor {
  private final SummationService summationService;
  DataProcessor(final SummationService summationService) {
    this.summationService = summationService;
  }
  void recieve(final int x) {
    summationService.addNew(x);
    //Do other things
  }
  void processData() {
    int sum = summationService.findSum();
    System.out.println("Processing data with sum = "  + sum);
    //Do other things
  }
}
```

আমি শুধুমাত্র *SummationService* এর ব্যবহারটুকু দেখালাম, ডাটা প্রসেসর আরো অনেক কাজ হয়তো করে। এখন আমরা এই কোডটাকে রান করার ব্যবস্থা করি এবং টেস্ট করে দেখি:

```
public class ExampleApp {
    public static void main(String[] args) {
        final DataProcessor simpleProcessor = new DataProcessor(new SimpleSum());
        simpleProcessor.recieve(10);
        simpleProcessor.recieve(10);
        simpleProcessor.recieve(20);
        simpleProcessor.processData();
        final DataProcessor uniqueProcessor = new DataProcessor(new UniqueSum());
        uniqueProcessor.recieve(10);
        uniqueProcessor.recieve(10);
        uniqueProcessor.recieve(20);
        simpleProcessor.processData();
    }
}
```

এখন মজা দেখো, আমরা *DataProcessor* কে দুই ভাবে তৈরি করতে পারছি, *new SimpleSum()* প্যারামিটার ব্যবহার এবং *new UniqueSum()* প্যারামিটার ব্যবহার করে। কারণ *SimpleSum* এবং *UniqueSum* দুইটা ক্লাসই আসলে *SummationService* ইন্টারফেসকে ইম্প্লিমেন্ট করে। *DataProcessor* এর জানা দরকার নাই কোন ধরনের ইম্প্লিমেন্টেশন ব্যবহার হচ্ছে, সে শুধু জানে তার একটা ইন্টারফেস আছে যেটা *addNew* এবং *findSum* যেগুলো ব্যবহার করে নতুন ডাটা যোগ করা যায় এবং যোগফল বের করা যায়। এখন যার যেরকম দরকার সে সেরকম ইম্প্লিমেন্টেশন ব্যবহার করবে আর *summationService* তার polymorphic শক্তি ব্যবহার করে সেই রূপ ধারণ করবে। প্রয়োজন মত বিভিন্ন রূপ ধারণ করাই polymorphism এর শক্তি।

Polymorphism যে শুধুমাত্র ইন্টারফেসের ক্ষেত্রে প্রয়োজ্য সেটা না, আমরা পরবর্তিতে এটা নিয়ে আরো বিস্তারিত জানবো।

আরো পরিস্কার করে বোঝার জন্য বাস্তব একটা ব্যবহার দেখি। জাভাতে লিস্ট ব্যবহার করার দরকার হলে আমরা সাধারণ এরকম লিখি:

```
List<Integer> list = new ArrayList<>();
```

এখানে আসলে কি ঘটছে? List হলো জাভার অফিসিয়াল লাইব্রেরিতে ডিফাইন করা একটা ইন্টারফেস, এবং ArrayList তাদেরই করা একটা ইম্প্লিমেন্টশন। এখন মনে করো তোমার ArrayList এ কাজ হচ্ছে না, লিংকড লিস্ট দরকার কারণ অ্যারেতে এলিমেন্ট ডিলিট করতে সময় বেশি লাগে, তাহলে তুমি লিখতে পারো:

```
List<Integer> list = new LinkedList<>();
```

LinkedList ও আসলে List ইন্টারফেসের আরেক রকমের ইমপ্লিমেন্টেশন। list একটা পলিমর্ফিক অবজেক্ট, সেখানে ArrayList অ্যাসাইন করা যায়, LinkedList ও করা যায়। তুমি যদি একটা মেথড তৈরি করো যেটা List টাইপকে প্যারামিটার হিসাবে নেয় তাহলে সেই মেথডে তুমি যেকোনো অবজেক্ট পাঠাতে পারবে যেটা List কে ইম্প্লিমেন্ট করে, চেষ্টা করে দেখতে পারো।

বোয়িং ৭৩৭-ম্যাক্স ট্রাজেডি

তোমরা হয়তো কিছুদিন আগে বোয়িং এর ৭৩৭-ম্যাক্স সিরিজের দুটি উড়োজাহাজ ক্র্যাশ করেছে। এই প্লেনের সফটওয়্যার ডিজাইনে সমস্যা ছিলো। বোয়িং পুরো প্লেনের কিছু ফিচারের ইমপ্লিমেন্টেশন বদলে ফেলেছে কিন্তু তারা সেগুলোর ইন্টারফেসে কোনো আপডেট করতে চায় নি। তারা ভেবেছে ইন্টারফেস একই রাখলে পাইলটদের নতুন করে ট্রেনিং দেয়ার দরকার হবে না। এরপর যেটা ঘটলো, প্লেনের অটোপাইলট ফিচারের ইম্প্লিমেন্টেশনের ভজঘটের কারণে প্লেন আকাশ থেকে মাটির দিকে ধেয়ে যাওয়া শুরু করলো, পাইলটরা কিছুই বুঝতে পারলো না কি ঘটছে, তাদের কাছে এমন কোনো ইন্টারফেসও নেই যে কন্ট্রোল নিজের হাতে তুলে নিবে। দুটি প্লেনে শতশহ মানুষ মারা গেল এই ভুলের কারণে, বর্তমানে সব এয়ারলাইন্স প্লেনটা বর্জন করেছে। সফটওয়্যার বাগ ভুল কতটা মারাত্মক হতে পারে এর থেকে বড় উদাহরণ আর কি হতে পারে?

তাহলে আজকে আমরা শিখলাম:

- Interface কি?
- Abstract Method কি?
- Polymorphism কি?
- Interface ব্যবহার করে কিভাবে Polymorphism অর্জন করা যায়?

তোমার কাজ হবে প্রতিটি কিওয়ার্ড গুগলে সার্চ করে বিস্তারিত শিখে ফেলা। ওরাকলে অফিসিয়াল ওয়েবসাইটে সহজ করে সবকিছু বুঝিয়ে দেয়া আছে, সেগুলো পড়ে ফেলতে পারো।

পরবর্তি পর্বে Inheritence কাকে বলে এবং এর ব্যবহার নিয়ে জানবো। আজ এই পর্যন্তই, কোনো অংশ বুঝতে সমস্যা হলে কমেন্ট অংশে জানাতে পারো।