জাভা প্রোগ্রামিং কোর্স
JAVA PROGRAMMING COURSE

লেখকঃ
মুহাম্মাদ মুহসিন

জাভা প্রোগ্রামিং কোর্স

Java Programming Course

লেখকঃ
মুহাম্মাদ মুহসিন

প্রথম প্রকাশ
পহেলা ডিসেম্বর ২০২৪

সতর্কতা
এই বইটি বিক্রয়ের জন্য নয়
বইটি বিনামূল্যে বিতরণযোগ্য

লেখকের কিছু কথা

আপনি কি জানেন, প্রোগ্রামিং শিখতে গিয়ে অনেকেরই প্রথম চ্যালেঞ্জ আসে কোথা থেকে শুরু করবেন তা বুঝতে না পারা! তবে চিন্তা করার কিছু নেই। আপনি যে কোন স্তরের শিক্ষার্থী হোন না কেন, এই বইটি আপনাকে জাভা প্রোগ্রামিং ভাষা শিখতে সহজ এবং সোজা উপায়ে সাহায্য করবে।

এই বইটির প্রতিটি অধ্যায় আপনাকে A থেকে Z পর্যন্ত জাভা প্রোগ্রামিংয়ের মৌলিক ধারণা থেকে শুরু করে অ্যাডভান্সড টপিক পর্যন্ত সব কিছু শিখাবে। আপনি একে একে শিখবেন কীভাবে কম্পিউটারের সাথে কথা বলতে হয়, কোড লিখতে হয়, এবং কীভাবে সফটওয়্যার তৈরি করতে হয়। প্রতিটি টপিক খুব সহজ ভাষায় বোঝানো হয়েছে যাতে আপনার কোনও সমস্যা না হয়। বিশ্ববিদ্যালয়ের প্রথম বছর বা কর্মজীবনে নতুন প্রোগ্রামার যাই হন, এই বই আপনাকে প্রথম দিন থেকে শুরু করে প্রোগ্রামিংয়ের মাস্টারি করতে সহায়তা করবে।

আপনার যদি প্রোগ্রামিংয়ের কোন পূর্ব অভিজ্ঞতা না থাকে, তাও সমস্যা নেই। এই বইটি এমনভাবে লেখা হয়েছে যাতে আপনি ধাপে ধাপে প্রোগ্রামিংয়ের সহজ ও মৌলিক বিষয়গুলো শিখতে পারবেন। একে একে কোডিংয়ের সবচেয়ে মজাদার দিকগুলো খুঁজে পাবেন, এবং আপনি শিখবেন কীভাবে জাভার শক্তিশালী বৈশিষ্ট্য ব্যবহার করে আসল প্রকল্প তৈরি করতে হয়।

এখনই শুরু করুন, এবং দেখুন, কীভাবে আপনি একটি দক্ষ প্রোগ্রামার হতে পারেন, খুব সহজে এবং দ্রুত! এখনই আপনার সফটওয়্যার তৈরি করার যাত্রা শুরু করুন!

মুহাম্মাদ মুহসিন
টুঙ্গিপাড়া, গোপালগঞ্জ, ঢাকা, বাংলাদেশ

সূচিপত্র

জাভা প্রোগ্রামিং

জাভা (Java) একটি জনপ্রিয়, উচ্চ স্তরের, অবজেক্ট-ওরিয়েন্টেড প্রোগ্রামিং ভাষা যা ১৯৯৫ সালে সান মাইক্রোসিস্টেমস (বর্তমানে Oracle) দ্বারা প্রকাশিত হয়। এটি বহুমুখী এবং প্ল্যাটফর্ম স্বাধীন (platform-independent), যার অর্থ আপনি একবার কোড লিখে তা যেকোনো প্ল্যাটফর্মে চালাতে পারবেন (যেমন Windows, Linux, MacOS, ইত্যাদি)। এর প্রধান বৈশিষ্ট্যগুলির মধ্যে রয়েছে শক্তিশালী নিরাপত্তা, মাল্টিথ্রেডিং সমর্থন, এবং গারবেজ কালেকশন।

জাভা প্রোগ্রামিং এর কিছু মূল বৈশিষ্ট্য:

- **Object-Oriented:** জাভা একটি অবজেক্ট-ওরিয়েন্টেড ভাষা, যেখানে ডেটা এবং ফাংশনগুলো অবজেক্টের মধ্যে রাখা হয়।
- **Platform Independent:** জাভার কোড JVM (Java Virtual Machine)-এর মাধ্যমে রান করা হয়, যার ফলে কোড একবার লিখে যেকোনো প্ল্যাটফর্মে চালানো যায়।
- **Multithreading:** জাভা মাল্টিথ্রেডিং সমর্থন করে, যার মাধ্যমে একাধিক কাজ একযোগে চলতে পারে।
- **Garbage Collection:** জাভাতে গারবেজ কালেকশন থাকে, যা অপ্রয়োজনীয় মেমরি স্বয়ংক্রিয়ভাবে মুক্ত করে দেয়।

একটি সহজ জাভা প্রোগ্রাম উদাহরণ:

java
Copy code
```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

ব্যাখ্যা:

- public class HelloWorld: এটি একটি ক্লাস ডেফিনিশন। জাভাতে সমস্ত কোড ক্লাসের মধ্যে থাকতে হয়।
- public static void main(String[] args): এটি মূল মেথড যা জাভা প্রোগ্রামের এন্ট্রি পয়েন্ট। String[] args হলো কমান্ড লাইন আর্গুমেন্টগুলি।
- System.out.println("Hello, World!");: এটি স্ক্রীনে "Hello, World!" প্রিন্ট করবে।

জাভাতে কাজ করার জন্য কিছু সাধারণ পদক্ষেপ:

- **কোড লেখা**: একটি .java এক্সটেনশনে কোড লিখুন (যেমন HelloWorld.java).
- **কোম্পাইল করা**: টার্মিনাল বা কমান্ড প্রম্পটে javac HelloWorld.java কমান্ড ব্যবহার করে কোড কম্পাইল করুন।
- **রান করা**: কম্পাইল হওয়া .class ফাইলটি চালানোর জন্য java HelloWorld কমান্ড ব্যবহার করুন।

জাভার কিছু সাধারণ কনসেপ্ট:

- **ভ্যারিয়েবলস**: জাভাতে ভ্যারিয়েবল ডেক্লেয়ার করতে হয়, যেমন:

```
java
Copy code
int num = 10;
String name = "John";
```

- **লোপ (Loop)**: জাভাতে সাধারণত for, while, বা do-while লোপ ব্যবহার করা হয়।

```
java
Copy code
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

- **কন্ডিশন (Conditionals)**: if, else if, else ব্যবহার করে শর্তমূলক বিবৃতি লেখা হয়।

```
java
Copy code
if (num > 0) {
    System.out.println("Positive number");
} else {
    System.out.println("Negative number");
}
```

আপনি যদি আরো গভীরে জানতে চান বা কোনো নির্দিষ্ট বিষয়ের ওপর সাহায্য চান, যেমন ক্লাস, অবজেক্ট, এক্সেপশন হ্যান্ডলিং, বা মাল্টিথ্রেডিং, আমাকে জানাতে পারেন!

জাভা ভাষার পরিচিতি

জাভা (Java) একটি উচ্চ স্তরের, অবজেক্ট-ওরিয়েন্টেড, ক্লাস ভিত্তিক, সাধারণ উদ্দেশ্যপ্রণোদিত প্রোগ্রামিং ভাষা। এটি ১৯৯৫ সালে সান মাইক্রোসিস্টেমস (বর্তমানে Oracle Corporation) দ্বারা তৈরি এবং প্রকাশিত হয়। জাভা ভাষাটি তৈরি করেছিলেন **জেমস গস্লিং** এবং তার সহকর্মীরা।

জাভা ভাষার মূল বৈশিষ্ট্য হলো এর **"Write Once, Run Anywhere"** (WORA) ধারণা, অর্থাৎ একবার কোড লিখে তা যেকোনো প্ল্যাটফর্মে নির্বিঘ্নে চালানো সম্ভব। এটি **প্ল্যাটফর্ম স্বাধীন** (platform-independent) হওয়ার কারণে অনেক জনপ্রিয় এবং বহুল ব্যবহৃত ভাষা।

জাভার বৈশিষ্ট্য:

- **অবজেক্ট-ওরিয়েন্টেড প্রোগ্রামিং (OOP):**
  - জাভা একটি সম্পূর্ণ অবজেক্ট-ওরিয়েন্টেড ভাষা, যেখানে কোড এবং ডেটা একত্রে অবজেক্ট হিসেবে থাকে। এতে ক্লাস এবং অবজেক্টের মাধ্যমে পুনঃব্যবহারযোগ্য এবং মডুলার কোড লেখা যায়।
  - প্রধান OOP কনসেপ্টগুলো:
    - Encapsulation (ক্লাসের মাধ্যমে ডেটা এবং ফাংশন একত্রিত করা)
    - Inheritance (একটি ক্লাসের বৈশিষ্ট্য অন্য একটি ক্লাসে উত্তরাধিকারসূত্রে প্রাপ্ত)

- **Polymorphism** (একই নামের মেথড বিভিন্নভাবে কাজ করা)
- **Abstraction** (প্রয়োজনীয় বৈশিষ্ট্য গোপন রেখে ব্যবহারকারীর কাছে সহজ ইন্টারফেস প্রদান করা)

➢ **প্ল্যাটফর্ম স্বাধীনতা (Platform Independence):**

- জাভা কোডকে **Java Virtual Machine (JVM)** মাধ্যমে চালানো হয়, যা কোডকে যেকোনো প্ল্যাটফর্মে (Windows, Linux, macOS) চালানোর সুবিধা দেয়। এজন্য জাভাকে **WORA** (Write Once, Run Anywhere) বলা হয়।

➢ **গারবেজ কালেকশন (Garbage Collection):**

- জাভাতে মেমরি ব্যবস্থাপনা স্বয়ংক্রিয়ভাবে করা হয়। যখন কোনো অবজেক্ট আর ব্যবহৃত হয় না, তখন গারবেজ কালেকশন তা মুছে ফেলে, ফলে মেমরি লিকের সমস্যা কমে যায়।

➢ **মাল্টিথ্রেডিং (Multithreading):**

- জাভা মাল্টিথ্রেডিং সমর্থন করে, যার মাধ্যমে একাধিক কাজ একসঙ্গে চালানো যায়। এটি প্রধানত বড় আয়তনের অ্যাপ্লিকেশন এবং গেমসের ক্ষেত্রে অত্যন্ত গুরুত্বপূর্ণ।

➢ **সিরিয়ালাইজেশন (Serialization):**

- জাভা অবজেক্টকে বাইনারি ফরম্যাটে রূপান্তরিত করতে এবং সংরক্ষণ করতে সক্ষম, যার মাধ্যমে অবজেক্টগুলি নেটওয়ার্কের মাধ্যমে পাঠানো বা ডাটাবেসে সেভ করা যায়।

➢ **নিরাপত্তা (Security):**

- জাভাতে বিল্ট-ইন নিরাপত্তা বৈশিষ্ট্য রয়েছে, যেমন **bytecode verification, sandboxing** এবং **cryptography**। এর ফলে এটি একটি নিরাপদ প্ল্যাটফর্ম হিসেবে ব্যবহৃত হয়।

➢ **সহজ এবং শক্তিশালী (Simple and Robust):**

- জাভা C এবং C++ এর তুলনায় অনেক বেশি সহজ এবং নিরাপদ। এর সিম্পল সিনট্যাক্স, মেমরি ম্যানেজমেন্ট এবং এক্সেপশন হ্যান্ডলিং কোডিংকে সহজ এবং এর বাগ ফিক্সিং সহজ করে তোলে।

জাভার ইতিহাস:

- **১৯৯১**: সান মাইক্রোসিস্টেমস শুরু করে একটি নতুন প্রোগ্রামিং ভাষার ডেভেলপমেন্ট যার কোডনেম ছিল **"Oak"**। এটি মূলত টেলিভিশন সেট-টপ বক্সে ব্যবহারের জন্য তৈরি করা হয়েছিল।
- **১৯৯৫**: "Oak" নামটি ট্রেডমার্ক সমস্যার কারণে পরিবর্তিত হয় এবং নতুন নাম হিসেবে **"Java"** রাখা হয়। তখন থেকেই জাভা একটি কমপ্লিট প্রোগ্রামিং ভাষা হিসেবে জনপ্রিয় হতে শুরু করে।
- **২০০৪**: জাভা ৫.০ (J2SE 5.0) রিলিজ করা হয়, যেখানে নতুন ফিচার যেমন **Generics, Metadata Annotations, Autoboxing, Enumerated Types** যুক্ত করা হয়।

জাভার ব্যবহার:

- **ওয়েব অ্যাপ্লিকেশন**: জাভা ব্যবহার করে ডাইনামিক ওয়েবসাইট এবং অ্যাপ্লিকেশন তৈরি করা যায় (যেমন: Servlets, JSP ইত্যাদি)।
- **মোবাইল অ্যাপ্লিকেশন**: Android অ্যাপ্লিকেশন ডেভেলপমেন্টের জন্য মূল ভাষা হিসেবে জাভা ব্যবহৃত হয়।
- **ডেস্কটপ অ্যাপ্লিকেশন**: জাভা GUI লাইব্রেরি যেমন **Swing** এবং **JavaFX** ব্যবহার করে ডেস্কটপ অ্যাপ্লিকেশন তৈরি করা যায়।
- **এন্টারপ্রাইজ অ্যাপ্লিকেশন**: জাভা EE (Enterprise Edition) প্ল্যাটফর্ম ব্যবহার করে বড় আকারের এন্টারপ্রাইজ সলিউশন তৈরি করা হয়।
- **বিগ ডেটা ও ক্লাউড কম্পিউটিং**: জাভা অনেক বিগ ডেটা ফ্রেমওয়ার্ক যেমন **Hadoop** এবং **Apache Spark** এর সাথে ব্যবহার হয়।

জাভার সুবিধাসমূহ:

- **প্ল্যাটফর্ম স্বাধীনতা**: একবার লেখা কোড যেকোনো অপারেটিং সিস্টেমে চালানো যায়।
- **অন্তর্নিহিত নিরাপত্তা**: জাভাতে নিরাপত্তা ফিচারসহ একটি শক্তিশালী নিরাপত্তা মডেল রয়েছে।
- **উচ্চ পারফরম্যান্স**: জাভা কন্টেনারাইজড অ্যাপ্লিকেশন এবং ক্লাউড পরিবেশে উচ্চ পারফরম্যান্স দিতে সক্ষম।

- **বিশাল লাইব্রেরি:** জাভার বিশাল স্ট্যান্ডার্ড লাইব্রেরি এবং APIs রয়েছে, যা বিভিন্ন কাজের জন্য ব্যবহার করা যায়।

জাভার কিছু জনপ্রিয় ফ্রেমওয়ার্ক:

- **Spring:** জাভা এন্টারপ্রাইজ অ্যাপ্লিকেশন ডেভেলপমেন্টের জন্য জনপ্রিয় একটি ফ্রেমওয়ার্ক।
- **Hibernate:** জাভা থেকে ডাটাবেসের সাথে যোগাযোগ সহজ করতে একটি ORM ফ্রেমওয়ার্ক।
- **Apache Struts:** ওয়েব অ্যাপ্লিকেশন ডেভেলপমেন্টের জন্য একটি ফ্রেমওয়ার্ক।

জাভা একটি শক্তিশালী, বহুমুখী এবং নিরাপদ প্রোগ্রামিং ভাষা যা বিভিন্ন ধরনের অ্যাপ্লিকেশন তৈরি করতে ব্যবহৃত হয়। এটি ডেভেলপারদের জন্য সহজ, কার্যকরী এবং স্কেলেবল কোড লেখার সুযোগ প্রদান করে। জাভা একে অপরের সাথে একাধিক প্ল্যাটফর্মে কাজ করতে সক্ষম হওয়ার কারণে এটি বিভিন্ন ডোমেনে ব্যাপকভাবে ব্যবহৃত হয়, বিশেষ করে বড় আকারের এন্টারপ্রাইজ সিস্টেম, ওয়েব অ্যাপ্লিকেশন এবং মোবাইল অ্যাপ্লিকেশনে।

আপনি যদি জাভার কোনো নির্দিষ্ট দিক নিয়ে আরও জানতে চান, যেমন এর সিনট্যাক্স, ফিচার, অথবা ব্যবহার, তাহলে আমাকে জানাতে পারেন!

জাভা সিনট্যাক্স (Java Syntax)

জাভার সিনট্যাক্স সেই নিয়ম এবং গঠন যা জাভা কোড লেখার সময় অনুসরণ করতে হয়। এটি প্রোগ্রামিং ভাষার মৌলিক কাঠামো, যা কোডের কার্যকারিতা এবং পাঠযোগ্যতা নিশ্চিত করে।

১. ক্লাস (Class)

জাভাতে সবকিছু একটি ক্লাসের মধ্যে থাকে। একটি প্রোগ্রামের মূল ক্লাস ডিফাইন করতে হয়, এবং তার মধ্যে main মেথড থাকতে হবে, যেখান থেকে প্রোগ্রাম execution শুরু হয়।

```
java
Copy code
public class HelloWorld {
    public static void main(String[] args) {
```

```
        System.out.println("Hello, World!");
    }
}
```

- public class HelloWorld: এটি ক্লাসের ডিফিনিশন। এখানে HelloWorld ক্লাসের নাম।
- public static void main(String[] args): এটি প্রোগ্রামের মেন মেথড। main মেথডে কোড লিখলেই তা রান হবে।

২. ভ্যারিয়েবল (Variables)

জাভাতে ভ্যারিয়েবল ঘোষণা করতে প্রথমে তার ডেটা টাইপ এবং তারপর ভ্যারিয়েবলের নাম লিখতে হয়।

```
java
Copy code
int age = 25;           // integer type variable
String name = "John";  // string type variable
boolean isStudent = true;  // boolean type variable
```

৩. ডেটা টাইপ (Data Types)

জাভাতে দুই ধরনের ডেটা টাইপ রয়েছে: **প্রিমিটিভ টাইপ** এবং **রেফারেন্স টাইপ**।

৩.১. প্রিমিটিভ ডেটা টাইপ:

- **int** – পূর্ণসংখ্যা (Integer)
  - উদাহরণ: int age = 25;
- **double** – দশমিক মান (Floating point)
  - উদাহরণ: double weight = 65.5;
- **char** – একক ক্যারেকটার
  - উদাহরণ: char grade = 'A';

- **boolean** – সত্য (true) বা মিথ্যা (false) মান
  - উদাহরণ: boolean isStudent = true;
- **byte** – 8-বিট পূর্ণসংখ্যা
  - উদাহরণ: byte b = 100;
- **short** – 16-বিট পূর্ণসংখ্যা
  - উদাহরণ: short s = 5000;
- **long** – 64-বিট পূর্ণসংখ্যা
  - উদাহরণ: long distance = 150000L;
- **float** – 32-বিট দশমিক মান
  - উদাহরণ: float pi = 3.14f;

৩.২. রেফারেন্স ডেটা টাইপ:

- **String** – শব্দের সিরিজ
  - উদাহরণ: String name = "John";
- **Array** – একাধিক একই ধরনের মান রাখে
  - উদাহরণ: int[] numbers = {1, 2, 3, 4};
- **Class/Object** – কাস্টম ডেটা টাইপ
  - উদাহরণ: Student student1 = new Student();

৪. কন্ট্রোল স্ট্রাকচার (Control Structures)

জাভাতে কন্ট্রোল স্ট্রাকচারগুলি কোডের কার্যপ্রবাহ নিয়ন্ত্রণ করতে ব্যবহৃত হয়। এটি সাধারণত **শর্তমূলক স্টেটমেন্ট** এবং **লুপ** এর মাধ্যমে কাজ করে।

৪.১. শর্তমূলক স্টেটমেন্ট (Conditional Statements)

- **if statement:**
  - if শর্তের ভিত্তিতে কোনো কোড এক্সিকিউট করে।

```
java
Copy code
int age = 20;
if (age >= 18) {
    System.out.println("Adult");
}
```

- **if-else statement:**
  - যদি শর্ত সত্য হয়, তবে একটি কোড এক্সিকিউট হবে, আর না হলে অন্য একটি কোড এক্সিকিউট হবে।

```
java
Copy code
int age = 16;
if (age >= 18) {
    System.out.println("Adult");
} else {
    System.out.println("Not an adult");
}
```

- **else-if ladder:**

- একাধিক শর্ত পরীক্ষা করতে else if ব্যবহার করা হয়।

```
java
Copy code
int marks = 75;
if (marks >= 90) {
    System.out.println("A+");
} else if (marks >= 75) {
    System.out.println("A");
} else {
    System.out.println("B");
}
```

- **switch statement:**

  - switch একাধিক শর্তের মধ্যে একটি ম্যাচ করার জন্য ব্যবহার করা হয়।

```
java
Copy code
int day = 3;
switch (day) {
    case 1:
        System.out.println("Sunday");
        break;
    case 2:
        System.out.println("Monday");
        break;
    case 3:
        System.out.println("Tuesday");
        break;
    default:
```

```
        System.out.println("Invalid day");
    }
```

৪.২. লুপ (Loops)

- **for loop:**
  - নির্দিষ্ট সংখ্যক বার কোড চালানোর জন্য ব্যবহার করা হয়।

```
java
Copy code
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

- **while loop:**
  - শর্ত সঠিক থাকলে কোড চালাতে থাকে।

```
java
Copy code
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

- **do-while loop:**
  - প্রথমে কোড এক্সিকিউট করে, তারপর শর্ত পরীক্ষা করে।

```
java
Copy code
int i = 0;
```

```
do {
    System.out.println(i);
    i++;
} while (i < 5);
```

৪.৩. ব্রেক এবং কন্টিনিউ (Break and Continue)

- **break**: লুপ বা সুইচ স্টেটমেন্ট থেকে দ্রুত বেরিয়ে আসতে ব্যবহৃত হয়।

```
java
Copy code
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break;  // Loop will terminate when i == 3
    }
    System.out.println(i);
}
```

- **continue**: লুপের বর্তমান iteration বাদ দিয়ে পরবর্তী iteration শুরু করতে ব্যবহৃত হয়।

```
java
Copy code
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;  // Skip the iteration when i == 3
    }
    System.out.println(i);
}
```

- **সিনট্যাক্স**: জাভার সিনট্যাক্স খুবই নির্দিষ্ট এবং নিয়ম অনুসরণ করতে হয়, যাতে কোডের কার্যকারিতা এবং পাঠযোগ্যতা বজায় থাকে।

- **ডেটা টাইপ:** প্রোগ্রাম লেখার সময় কোন ধরনের ডেটা ব্যবহার করা হবে তা ঠিক করতে ডেটা টাইপ গুরুত্বপূর্ণ ভূমিকা পালন করে।
- **কন্ট্রোল স্ট্রাকচার:** শর্ত, লুপ এবং অন্যান্য কন্ট্রোল স্ট্রাকচার ব্যবহারের মাধ্যমে কোডের কার্যপ্রবাহ নিয়ন্ত্রণ করা হয়।

আপনি যদি আরও বিস্তারিত জানাতে চান বা কিছু নির্দিষ্ট অংশ নিয়ে সাহায্য চান, তাহলে আমাকে জানাতে পারেন!

জাভাতে ক্লাস, অবজেক্ট এবং ইনহেরিটেন্স (Inheritance) - এর গুরুত্বপূর্ণ কনসেপ্ট

জাভা একটি **অবজেক্ট-ওরিয়েন্টেড প্রোগ্রামিং** (OOP) ভাষা, এবং এটি **ক্লাস**, **অবজেক্ট**, এবং **ইনহেরিটেন্স** এর মতো কনসেপ্টকে গুরুত্ব দিয়ে কাজ করে। এগুলোর মাধ্যমে কোডকে আরও মডুলার, পুনঃব্যবহারযোগ্য এবং সহজে রক্ষণাবেক্ষণযোগ্য করা সম্ভব হয়। নিচে এই কনসেপ্টগুলির ব্যাখ্যা দেওয়া হলো।

---

১. ক্লাস (Class)

একটি **ক্লাস** হল একটি ব্লুপ্রিন্ট বা টেমপ্লেট, যা অবজেক্ট তৈরি করার জন্য ব্যবহৃত হয়। ক্লাসে ডেটা (ফিল্ডস বা ভ্যারিয়েবলস) এবং মেথডস (ফাংশন) থাকে যা অবজেক্টের বৈশিষ্ট্য এবং আচরণ বর্ণনা করে।

- **ক্লাসের গঠন:**
    - **ফিল্ডস** (Fields): ডেটা সদস্য, যা অবজেক্টের বৈশিষ্ট্য বর্ণনা করে।
    - **মেথডস** (Methods): কোডের ব্লক যা নির্দিষ্ট কাজ সম্পন্ন করে (যেমন: ডেটা সেট করা বা গ্রহণ করা)।
    - **কনস্ট্রাক্টর** (Constructor): অবজেক্ট তৈরি করার জন্য ব্যবহৃত বিশেষ মেথড।

উদাহরণ: ক্লাস ডিফিনিশন

java
Copy code
public class Car {

```
    // ফিল্ডস (Fields)
    String brand;
    String model;
    int year;

    // কনস্ট্রাক্টর (Constructor)
    public Car(String brand, String model, int year) {
        this.brand = brand;
        this.model = model;
        this.year = year;
    }

    // মেথড (Method)
    public void displayInfo() {
        System.out.println("Car brand: " + brand);
        System.out.println("Car model: " + model);
        System.out.println("Car year: " + year);
    }
}
```

- এখানে Car ক্লাসের তিনটি ফিল্ড (brand, model, year) এবং একটি মেথড displayInfo() রয়েছে যা গাড়ির তথ্য প্রদর্শন করবে।
- Car(String brand, String model, int year) হলো কনস্ট্রাক্টর, যা গাড়ির তথ্য ইনিশিয়ালাইজ করবে যখন একটি নতুন অবজেক্ট তৈরি হবে।

---

২. অবজেক্ট (Object)

অবজেক্ট হলো একটি ক্লাসের একটি নির্দিষ্ট ইনস্ট্যান্স। ক্লাস একটি ব্লুপ্রিন্ট হলেও অবজেক্ট একটি বাস্তব উপাদান যা ক্লাসের সকল বৈশিষ্ট্য এবং আচরণ ধারণ করে।

অবজেক্ট তৈরি করতে হলে প্রথমে সেই ক্লাসের একটি ইনস্ট্যান্স তৈরি করতে হয়।

উদাহরণ: অবজেক্ট তৈরি ও ব্যবহার

```
java
Copy code
public class Main {
    public static void main(String[] args) {
        // একটি অবজেক্ট তৈরি করা হচ্ছে
        Car car1 = new Car("Toyota", "Corolla", 2020);

        // অবজেক্টের মেথড কল করা
        car1.displayInfo();
    }
}
```

- এখানে car1 হলো Car ক্লাসের একটি অবজেক্ট।
- new Car("Toyota", "Corolla", 2020) এর মাধ্যমে car1 অবজেক্ট তৈরি করা হয়েছে এবং এটি Car ক্লাসের কনস্ট্রাক্টর ব্যবহার করে গাড়ির ব্র্যান্ড, মডেল এবং বছর সেট করা হয়েছে।

আবস্ট্রাক্টলি, আপনি ভাবতে পারেন:

- **ক্লাস** হলো একটি নীলনকশা বা রূপরেখা।
- **অবজেক্ট** হলো সেই রূপরেখা অনুসারে বাস্তবে তৈরি কোনো বস্তু।

---

৩. ইনহেরিটেন্স (Inheritance)

ইনহেরিটেন্স হলো এক ক্লাসের বৈশিষ্ট্য এবং আচরণ অন্য একটি ক্লাসে উত্তরাধিকারসূত্রে প্রাপ্ত হওয়া। এটি কোড পুনঃব্যবহারযোগ্য করে তোলে এবং **বিশেষীকরণ** (specialization) এবং **অবস্ট্রাকশন** এর মাধ্যমে কোডের মডুলারিটি বাড়ায়।

ইনহেরিটেন্সের মূল ধারণা:

- একটি **সুপার ক্লাস** (Super Class) বা **প্যারেন্ট ক্লাস** হতে একটি **সাব ক্লাস** (Sub Class) বা **চাইল্ড ক্লাস** তার বৈশিষ্ট্য (ফিল্ডস) এবং আচরণ (মেথডস) উত্তরাধিকারসূত্রে প্রাপ্ত করে।

উদাহরণ: ইনহেরিটেন্স

java
Copy code

```
// সুপার ক্লাস (Super Class)
public class Animal {
    String name;

    public void eat() {
        System.out.println(name + " is eating.");
    }
}

// সাব ক্লাস (Sub Class)
public class Dog extends Animal {
    // নতুন মেথড (Method)
    public void bark() {
        System.out.println(name + " is barking.");
    }
}

public class Main {
    public static void main(String[] args) {
        // সাব ক্লাসের অবজেক্ট তৈরি করা
        Dog dog1 = new Dog();
        dog1.name = "Buddy";
```

```
        dog1.eat();  // সুপার ক্লাসের মেথড
        dog1.bark(); // সাব ক্লাসের মেথড
    }
}
```

ব্যাখ্যা:

- Animal ক্লাস হলো সুপার ক্লাস, যার মধ্যে একটি মেথড eat() এবং একটি ফিল্ড name রয়েছে।
- Dog ক্লাস হলো সাব ক্লাস, যা Animal ক্লাস থেকে ইনহেরিট করেছে এবং অতিরিক্ত একটি মেথড bark() যোগ করেছে।
- dog1.eat() সুপার ক্লাসের মেথড কল করছে এবং dog1.bark() সাব ক্লাসের মেথড কল করছে।

ইনহেরিটেন্সের সুবিধা:

- **কোড পুনঃব্যবহার:** সুপার ক্লাসের কোড সাব ক্লাসে পুনরায় লেখা প্রয়োজন হয় না।
- **পুনঃব্যবহারযোগ্যতা:** নতুন ক্লাস তৈরি করার সময় পূর্ববর্তী ক্লাসের কোড পুনঃব্যবহার করা যায়।
- **বিশেষীকরণ:** সাব ক্লাস নতুন আচরণ বা বৈশিষ্ট্য যোগ করে সুপার ক্লাসের আচরণে অতিরিক্ত কার্যকারিতা যোগ করতে পারে।

আরো কিছু বৈশিষ্ট্য:

- **super কিওয়ার্ড:** সাব ক্লাসের মধ্যে সুপার ক্লাসের কন্সট্রাক্টর বা মেথড/ফিল্ডে অ্যাক্সেস করতে super কিওয়ার্ড ব্যবহার করা হয়।

```
java
Copy code
public class Dog extends Animal {
    public Dog(String name) {
        super(name);  // সুপার ক্লাসের কন্সট্রাক্টর কল
    }
}
```

- **Method Overriding:** সাব ক্লাসে সুপার ক্লাসের মেথড আবার নতুনভাবে রিডিফাইন করা। এটি **ডাইনামিক পলিমরফিজম** হতে সাহায্য করে।

```
java
Copy code
public class Dog extends Animal {
    @Override
    public void eat() {
        System.out.println("Dog is eating bones.");
    }
}
```

উপসংহার

- **ক্লাস:** একটি টেমপ্লেট বা ব্লুপ্রিন্ট যা অবজেক্ট তৈরি করতে ব্যবহৃত হয়। ক্লাসে ফিল্ড (ভ্যারিয়েবল) এবং মেথড থাকে।
- **অবজেক্ট:** ক্লাসের একটি নির্দিষ্ট ইনস্ট্যান্স যা বাস্তব জগতের একটি বস্তু।
- **ইনহেরিটেন্স:** একটি ক্লাস অন্য ক্লাসের বৈশিষ্ট্য ও আচরণ গ্রহণ করে। এটি কোড পুনঃব্যবহার এবং শ্রেণীর মধ্যে সম্পর্ক তৈরি করতে সাহায্য করে।

এই কনসেপ্টগুলির মাধ্যমে জাভার অবজেক্ট-ওরিয়েন্টেড প্রোগ্রামিং শক্তিশালী এবং দক্ষ হয়ে ওঠে।

পলিমরফিজম, অ্যাবস্ট্র্যাকশন

১. পলিমরফিজম (Polymorphism)

পলিমরফিজম একটি গুরুত্বপূর্ণ **অবজেক্ট-ওরিয়েন্টেড প্রোগ্রামিং (OOP)** ধারণা, যা একাধিক রূপে আচরণ প্রদর্শন করার ক্ষমতাকে বোঝায়। এর মানে হল যে, একটি একক মেথড বা একটি অবজেক্ট ভিন্নভাবে কাজ করতে পারে বিভিন্ন কনটেক্সটে।

পলিমরফিজম দুটি প্রধান ধরনের হয়:

- **মেথড ওভারলোডিং (Method Overloading)**
- **মেথড ওভাররাইডিং (Method Overriding)**

১.১. মেথড ওভারলোডিং (Method Overloading)

মেথড ওভারলোডিং হল একই নামের একাধিক মেথড তৈরি করা, যেখানে প্রতিটি মেথডের প্যারামিটার সংখ্যা বা টাইপ আলাদা থাকে। এটি কোডের পুনঃব্যবহারযোগ্যতা বাড়ায়।

উদাহরণ:

```
java
Copy code
class Printer {
    // মেথড ওভারলোডিং
    public void print(int num) {
        System.out.println("Printing number: " + num);
    }

    public void print(String text) {
        System.out.println("Printing text: " + text);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer p = new Printer();
        p.print(123);          // এটি print(int num) মেথড কল করবে
        p.print("Hello!");     // এটি print(String text) মেথড কল করবে
    }
}
```

এখানে, Printer ক্লাসের দুটি print() মেথড রয়েছে: একটি int আর্গুমেন্ট নেয় এবং অন্যটি String আর্গুমেন্ট নেয়। এটি মেথড ওভারলোডিংয়ের উদাহরণ, যেখানে একই মেথড নাম ব্যবহার করা হয়েছে তবে প্যারামিটার ভিন্ন।

১.২. মেথড ওভাররাইডিং (Method Overriding)

মেথড ওভাররাইডিং হল সাব ক্লাসে সুপার ক্লাসের মেথড আবার নতুনভাবে সংজ্ঞায়িত করা। এটি **ডাইনামিক পলিমরফিজম** তৈরি করে, যেখানে একটি একক মেথড বিভিন্ন ক্লাসে আলাদা ভাবে কাজ করতে পারে।

উদাহরণ:

```
java
Copy code
// সুপার ক্লাস
class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}

// সাব ক্লাস
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
```

```
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog();
        Animal animal2 = new Cat();

        animal1.sound();  // Dog barks
        animal2.sound();  // Cat meows
    }
}
```

এখানে, Dog এবং Cat ক্লাস Animal ক্লাসের sound() মেথডটি ওভাররাইড করেছে। যখন animal1 এবং animal2 মেথড কল করা হয়, তখন নির্দিষ্ট ক্লাসের কাস্টম sound() মেথড রান হয়, যা **ডাইনামিক পলিমরফিজম** তৈরি করে। এই ধরনের পলিমরফিজম **রানটাইম পলিমরফিজম** হিসেবেও পরিচিত।

---

২. অ্যাবস্ট্র্যাকশন (Abstraction)

অ্যাবস্ট্র্যাকশন হল একটি প্রক্রিয়া যেখানে অবাস্তব বা অপ্রয়োজনীয় ডিটেইলস বাদ দিয়ে শুধুমাত্র গুরুত্বপূর্ণ এবং প্রয়োজনীয় তথ্য প্রদর্শন করা হয়। এটি প্রোগ্রামিংয়ের জটিলতাকে লুকিয়ে রাখে এবং ব্যবহারকারীর জন্য সহজতর ইন্টারফেস প্রদান করে।

অ্যাবস্ট্র্যাকশন অর্জন করা যায় দুটি উপায়ে:

- **অ্যাবস্ট্র্যাক্ট ক্লাস** (Abstract Class)
- **ইন্টারফেস** (Interface)

২.১. অ্যাবস্ট্র্যাক্ট ক্লাস (Abstract Class)

একটি **অ্যাবস্ট্র্যাক্ট ক্লাস** হলো এমন একটি ক্লাস যা সম্পূর্ণভাবে বাস্তবায়িত না হয়ে শুধুমাত্র অল্প কিছু মেথডের বেসিক কাঠামো প্রদান করে। অ্যাবস্ট্র্যাক্ট ক্লাসে একটি বা একাধিক **অ্যাবস্ট্র্যাক্ট মেথড** থাকতে পারে, যার মধ্যে কোন কনক্রিট (পুরোপুরি বাস্তবায়িত) কোড থাকে না। সাব ক্লাসে গিয়ে সেই মেথডগুলো সম্পূর্ণ করা হয়।

উদাহরণ:

```
java
Copy code
// অ্যাবস্ট্র্যাক্ট ক্লাস
abstract class Animal {
    String name;

    // অ্যাবস্ট্র্যাক্ট মেথড
    public abstract void sound();

    // কনক্রিট মেথড
    public void eat() {
        System.out.println(name + " is eating");
    }
}

// সাব ক্লাস
class Dog extends Animal {
    public Dog(String name) {
        this.name = name;
    }

    // অ্যাবস্ট্র্যাক্ট মেথড বাস্তবায়ন
```

```
    @Override
    public void sound() {
        System.out.println(name + " barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog("Buddy");
        dog.eat();  // কনক্রিট মেথড
        dog.sound(); // অ্যাবস্ট্র্যাক্ট মেথড
    }
}
```

এখানে, Animal একটি অ্যাবস্ট্র্যাক্ট ক্লাস এবং তার মধ্যে একটি অ্যাবস্ট্র্যাক্ট মেথড sound() রয়েছে। Dog ক্লাসে sound() মেথডটি সম্পূর্ণ বাস্তবায়িত হয়েছে।

২.২. ইন্টারফেস (Interface)

ইন্টারফেস হল একটি সম্পূর্ণ অ্যাবস্ট্র্যাক্ট ক্লাস যা শুধুমাত্র মেথড সিগনেচার (signature) প্রদান করে, কিন্তু কোনো কনক্রিট কোড থাকে না। একটি ক্লাস একাধিক ইন্টারফেস ইমপ্লিমেন্ট করতে পারে।

উদাহরণ:

```
java
Copy code
// ইন্টারফেস
interface Animal {
    void sound();
}

// ক্লাস যা ইন্টারফেস ইমপ্লিমেন্ট করছে
```

```
class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat implements Animal {
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();

        dog.sound();  // Dog barks
        cat.sound();  // Cat meows
    }
}
```

এখানে, Animal একটি ইন্টারফেস যা sound() মেথডের সিগনেচার রাখে। Dog এবং Cat ক্লাস এই ইন্টারফেসটি ইমপ্লিমেন্ট করেছে এবং তাদের নিজস্ব sound() মেথড বাস্তবায়ন করেছে।

পলিমরফিজম ও অ্যাবস্ট্র্যাকশনের মধ্যে পার্থক্য:

- **পলিমরফিজম:** একটি মেথড বা অবজেক্ট বিভিন্ন কনটেক্সটে ভিন্নভাবে আচরণ করতে পারে।
- **অ্যাবস্ট্র্যাকশন:** শুধুমাত্র গুরুত্বপূর্ণ তথ্য প্রকাশ করে এবং অবাস্তব বা অপ্রয়োজনীয় ডিটেইলস লুকিয়ে রাখে।

- **পলিমরফিজম** কোডের পুনঃব্যবহারযোগ্যতা এবং গতিশীল আচরণের সুবিধা প্রদান করে। এটি মেথড ওভারলোডিং এবং মেথড ওভাররাইডিং এর মাধ্যমে অর্জিত হয়।
- **অ্যাবস্ট্র্যাকশন** জটিলতা লুকিয়ে রাখে এবং শুধুমাত্র প্রয়োজনীয় তথ্য দেখায়, যা অ্যাবস্ট্র্যাক্ট ক্লাস বা ইন্টারফেসের মাধ্যমে অর্জিত হয়।

এই ধারণাগুলি সফটওয়্যার ডিজাইনকে আরো মডুলার, সোজা এবং পুনঃব্যবহারযোগ্য করে তোলে।

১. ইন্টারফেস (Interface)

ইন্টারফেস হলো একটি সম্পূর্ণ অ্যাবস্ট্র্যাক্ট ক্লাস, যেখানে শুধুমাত্র মেথড সিগনেচার (method signature) সংজ্ঞায়িত করা হয়, কিন্তু মেথডের বাস্তবায়ন (implementation) করা হয় না। এর মাধ্যমে আমরা একাধিক শ্রেণীর মধ্যে সাধারণ আচরণ নির্ধারণ করতে পারি।

একটি ক্লাস একাধিক ইন্টারফেস ইমপ্লিমেন্ট (implement) করতে পারে, যা **মাল্টিপল ইনহেরিটেন্স**-এর মতো আচরণ তৈরি করতে সাহায্য করে (যদিও জাভায় সরাসরি মাল্টিপল ইনহেরিটেন্স সম্ভব নয়)। ইন্টারফেসের মেথডগুলি public এবং abstract হয়ে থাকে (যেহেতু জাভাতে সব ইন্টারফেস মেথড স্বাভাবিকভাবে অ্যাবস্ট্র্যাক্ট হয়), এবং এগুলোকে ক্লাসে ইমপ্লিমেন্ট করা হয়।

ইন্টারফেসের মূল বৈশিষ্ট্য:

- সমস্ত মেথড abstract এবং public হয়।
- একটি ক্লাস একাধিক ইন্টারফেস ইমপ্লিমেন্ট করতে পারে।
- ইন্টারফেসের মধ্যে **ফিল্ড** কেবল static এবং final হতে পারে।
- ইন্টারফেসের মেথডগুলির বাস্তবায়ন অবশ্যই সাব ক্লাসে করতে হয়।

উদাহরণ:

java
Copy code
// ইন্টারফেস
interface Animal {

```
    void sound();  // abstract method (no body)
}

// ক্লাস যা ইন্টারফেস ইমপ্লিমেন্ট করেছে
class Dog implements Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

class Cat implements Animal {
    @Override
    public void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog();
        Animal myCat = new Cat();

        myDog.sound();  // Dog barks
        myCat.sound();  // Cat meows
    }
}
```

ব্যাখ্যা:

- এখানে Animal একটি ইন্টারফেস, যেটি sound() নামের একটি মেথড ডিফাইন করে।

- Dog এবং Cat ক্লাস দুটি Animal ইন্টারফেস ইমপ্লিমেন্ট করে এবং তাদের নিজস্ব sound() মেথডে আচরণ কাস্টমাইজ করে।

ইন্টারফেসের আরো কিছু সুবিধা:

- ❖ **Multiple Inheritance:** একাধিক ইন্টারফেস ইমপ্লিমেন্ট করার মাধ্যমে মাল্টিপল ইনহেরিটেন্সের সুবিধা পাওয়া যায়, যা জাভায় সরাসরি সম্ভব নয়।
- ❖ **Loose Coupling:** ইন্টারফেসের মাধ্যমে কোডে loose coupling (কম সম্পর্ক) বজায় রাখা সম্ভব, কারণ ইন্টারফেস শুধুমাত্র আচরণের চুক্তি প্রদান করে, বাস্তবায়ন করে না।

---

২. এক্সেপশন হ্যান্ডলিং (Exception Handling)

এক্সেপশন হ্যান্ডলিং একটি গুরুত্বপূর্ণ কৌশল যা প্রোগ্রামে যেকোনো ধরনের ত্রুটি (error) বা অসুবিধা (exception) মোকাবেলা করতে সাহায্য করে। জাভা প্রোগ্রামিংয়ে এক্সেপশন হলো এমন একটি ঘটনা যা প্রোগ্রামের স্বাভাবিক প্রবাহ ব্যাহত করে। এক্সেপশন হ্যান্ডলিং আমাদের সঠিকভাবে ত্রুটি হ্যান্ডেল করতে এবং প্রোগ্রামের ক্র্যাশ হওয়া প্রতিরোধ করতে সাহায্য করে।

এক্সেপশন হ্যান্ডলিং এর উপাদান:

- ➢ **Try Block:** কোডের সেই অংশ যা ত্রুটিপূর্ণ হতে পারে, সেখানেই try ব্লক ব্যবহার করা হয়।
- ➢ **Catch Block:** try ব্লকে কোনো ত্রুটি ঘটলে সেটি catch ব্লকে ধরা হয় এবং সেই ত্রুটির জন্য উপযুক্ত ব্যবস্থা নেয়া হয়।
- ➢ **Finally Block:** এটি একটি ঐচ্ছিক ব্লক যা try বা catch ব্লক থেকে নির্বিশেষে execute হয়। সাধারণত, finally ব্লককে ব্যবহার করা হয় রিসোর্স (যেমন, ফাইল, ডাটাবেস) বন্ধ করার জন্য।

এক্সেপশন হ্যান্ডলিং এর সাধারণ গঠন:

java
Copy code
```
try {
    // কোড যেখানে ত্রুটি ঘটতে পারে
```

```
} catch (ExceptionType1 e1) {
    // ত্রুটি হ্যান্ডলিং কোড
} catch (ExceptionType2 e2) {
    // অন্য ধরনের ত্রুটি হ্যান্ডলিং কোড
} finally {
    // এই ব্লকটি সবসময় execute হবে
}
```

উদাহরণ:

```
java
Copy code
public class ExceptionHandlingExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;  // এটি এক্সেপশন তৈরি করবে (division by zero)
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        } finally {
            System.out.println("This will always execute.");
        }
    }
}
```

ব্যাখ্যা:

- **try ব্লক**: এখানে একটি বিভাজন দ্বারা শূন্যে ভাগ করার চেষ্টা করা হয়েছে, যা ArithmeticException তৈরি করবে।
- **catch ব্লক**: এই ব্লকে ArithmeticException ধরা হয়েছে এবং ত্রুটির মেসেজ প্রিন্ট করা হয়েছে।
- **finally ব্লক**: এটি সবসময় রান হবে, এমনকি যদি কোনো ত্রুটি ঘটে থাকে বা না ঘটে।

এক্সেপশন হ্যান্ডলিংয়ের কিছু গুরুত্বপূর্ণ দিক:

- **কাস্টম এক্সেপশন (Custom Exception):** আপনি নিজেও কাস্টম এক্সেপশন তৈরি করতে পারেন যেগুলি আপনার প্রোগ্রামের জন্য নির্দিষ্ট ত্রুটির শর্তে কাজ করবে। এটি করতে Exception ক্লাসকে ইনহেরিট করতে হয়।

  উদাহরণ:

```
java
Copy code
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class Main {
    public static void validateAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or older");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(15);
        } catch (InvalidAgeException e) {
            System.out.println(e.getMessage());
        }
    }
```

}

- এখানে, InvalidAgeException একটি কাস্টম এক্সেপশন যা Exception ক্লাসকে ইনহেরিট করেছে।
- validateAge মেথডটি ১৮ বছরের কম বয়স থাকলে একটি InvalidAgeException থ্রো করবে এবং catch ব্লক তা হ্যান্ডল করবে।

➢ **throw এবং throws কিওয়ার্ড:**

- **throw:** একটি এক্সেপশন থ্রো করার জন্য ব্যবহার হয়। এটি মেথডের ভিতরেই ব্যবহার করা হয়।
- **throws:** মেথড সিগনেচারে ব্যবহার করা হয় যা নির্দেশ করে যে মেথডটি এক্সেপশন ছুঁড়ে ফেলতে পারে এবং সেই এক্সেপশনকে হ্যান্ডল করতে হবে।

উদাহরণ:

```
java
Copy code
public class Main {
    public static void main(String[] args) {
        try {
            divide(10, 0);  // Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

    public static void divide(int a, int b) throws ArithmeticException {
        if (b == 0) {
            throw new ArithmeticException("Cannot divide by zero");
        }
        System.out.println(a / b);
```

```
    }
}
```

- এখানে, divide মেথডে throws ArithmeticException যুক্ত করা হয়েছে, যা নির্দেশ করে যে divide মেথড একটি ArithmeticException থ্রো করতে পারে।

উপসংহার

- **ইন্টারফেস:**
  - একটি ইন্টারফেস হল একটি চুক্তি যা কোনো ক্লাসের জন্য একটি আচরণ নির্ধারণ করে, কিন্তু বাস্তবায়ন ক্লাসে করতে হয়।
  - একাধিক ইন্টারফেস ইমপ্লিমেন্ট করার মাধ্যমে কোডে মাল্টিপল ইনহেরিটেন্সের সুবিধা পাওয়া যায়।
- **এক্সেপশন হ্যান্ডলিং:**
  - এক্সেপশন হ্যান্ডলিং প্রোগ্রামের ত্রুটির সাথে সঠিকভাবে মোকাবেলা করার কৌশল।
  - try, catch, finally, throw, এবং throws কিওয়ার্ড ব্যবহার করে ত্রুটি সঠিকভাবে হ্যান্ডল করা হয়।
  - কাস্টম এক্সেপশন তৈরি করার মাধ্যমে বিশেষ ধরণের ত্রুটিও হ্যান্ডল করা সম্ভব।

১. মুলতুবি থ্রেডিং (Multithreading)

মুলতুবি থ্রেডিং (Multithreading) হল একটি প্রোগ্রামিং কৌশল যেখানে একটি প্রোগ্রাম একাধিক থ্রেডের মাধ্যমে একযোগে কাজ করে। একটি **থ্রেড** হল একটি **প্রসেসের ভিতরে কার্যকরী একক**। এটি প্রোগ্রামের কার্যক্রমের একটি পৃথক ধারার মতো কাজ করে। একাধিক থ্রেড ব্যবহার করে একসাথে কাজ সম্পাদন করা হয়, যার ফলে কোড আরও কার্যকর এবং দ্রুত হয়, বিশেষত এমন পরিস্থিতিতে যেখানে একাধিক কাজ একসঙ্গে সম্পাদিত হতে পারে (যেমন: ডেটা প্রসেসিং, ইউজার ইন্টারফেসের প্রতিক্রিয়া ইত্যাদি)।

থ্রেড কি?

থ্রেড হলো একটি কার্যকরী ইউনিট যা একটি প্রসেসের মধ্যে চালিত হয়। প্রতিটি থ্রেড একটি প্রোগ্রামের এক্সিকিউশন প্রভাবিত করে এবং একটি প্রসেসের অংশ হিসেবে কাজ করে।

একটি থ্রেডের প্রধান বৈশিষ্ট্য:

- **প্রসেসের মধ্যে কাজ:** একটি প্রসেসের মধ্যে একাধিক থ্রেড থাকতে পারে।
- **শেয়ারিং রিসোর্স:** একই প্রসেসের সব থ্রেড সাধারণভাবে একই মেমরি স্পেস শেয়ার করে।
- **স্বতন্ত্র এক্সিকিউশন:** প্রতিটি থ্রেড একে অপরের থেকে স্বাধীনভাবে কাজ করতে পারে, তবে তারা একই প্রসেসের অংশ হওয়ায় তাদের কিছু রিসোর্স শেয়ার করা হয়।

মুলতুবি থ্রেডিংয়ের সুবিধা:

- **দ্রুত পারফরম্যান্স:** একাধিক থ্রেড সমান্তরালভাবে কাজ করে একাধিক কাজ সম্পাদন করতে পারে, যার ফলে সিস্টেমের কাজের গতি বাড়ে।
- **বেটার ইউজার এক্সপেরিয়েন্স:** একাধিক থ্রেডের মাধ্যমে ইউজার ইন্টারফেস (UI) রেসপন্সিভ থাকে, যেমন ব্যাকগ্রাউন্ডে কাজ চলাকালে UI দ্রুত রেন্ডার হয়।
- **প্রসেসিং ক্ষমতা বৃদ্ধি:** মাল্টিপ্ল কোর প্রসেসরের সুবিধা নিয়ে থ্রেডিং মাধ্যমে সিস্টেমের শক্তি ব্যবহারের ক্ষমতা বৃদ্ধি পায়।

জাভাতে মুলতুবি থ্রেডিং

জাভাতে থ্রেডিং দুইভাবে করা যায়:

- **Thread ক্লাসের মাধ্যমে**
- **Runnable ইন্টারফেসের মাধ্যমে**

উদাহরণ ১: Thread ক্লাস ব্যবহার

java
Copy code

```
class MyThread extends Thread {
    public void run() {
```

```
        System.out.println("Thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Thread starts here
    }
}
```

- Thread ক্লাসে run() মেথডটি ওভাররাইড করা হয়, এবং start() মেথডটি থ্রেড চালু করার জন্য ব্যবহার করা হয়।

উদাহরণ ২: Runnable ইন্টারফেস ব্যবহার

```
java
Copy code
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Runnable thread is running");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable r1 = new MyRunnable();
        Thread t1 = new Thread(r1);
        t1.start();  // Thread starts here
    }
}
```

- এখানে Runnable ইন্টারফেস ইমপ্লিমেন্ট করা হয়েছে এবং Thread ক্লাসের মাধ্যমে Runnable ইন্টারফেসের run() মেথড চালানো হয়েছে।

---

২. কনকারেন্সি (Concurrency)

কনকারেন্সি (Concurrency) হল একাধিক কাজ একযোগে প্রক্রিয়া করার ক্ষমতা, যদিও তা একসঙ্গে না হয়ে সময় ভাগ করে (time-sharing) করা হয়। কনকারেন্সি যখন কার্যকরী, তখন একাধিক কাজ একটি সময়ের মধ্যে সম্পাদিত হতে পারে, কিন্তু আসলে এগুলি একে অপরের সাথে **শেয়ার করা** CPU সময় ব্যবহার করে।

কনকারেন্সির বৈশিষ্ট্য:

- কনকারেন্সি বাস্তবায়ন করতে থ্রেড ব্যবহার করা হয়, যেখানে প্রতিটি থ্রেড একটি আলাদা কাজ করে।
- থ্রেডের মধ্যে কনকারেন্সি ব্যবস্থাপনা **সিনক্রোনাইজেশন** দ্বারা করা হয়, যাতে একটি থ্রেডের কাজ অন্য থ্রেডের কাজের সাথে সংঘর্ষ না করে।
- **কনকারেন্সি** এবং **পারালালিজম** দুটি আলাদা ধারণা হলেও অনেক সময় একে অপরের সাথে সম্পর্কিত। কনকারেন্সি একসাথে একাধিক কাজ করার ধারণা, কিন্তু পারালালিজমে সব কাজ একসাথে বাস্তবায়িত হয় (যেমন মাল্টিপল প্রসেসর ব্যবহার করে)।

কনকারেন্সির সুবিধা:

- রেসপন্সিভ সিস্টেম: ইউজার ইন্টারফেস বা অন্যান্য আই/ও অপারেশন একাধিক থ্রেডে ভাগ করা গেলে সিস্টেমের রেসপন্সিভনেস উন্নত হয়।
- অপারেশনাল অ্যাডভান্টেজ: একাধিক কাজ সম্পাদন করতে একই সময় ব্যবহৃত হতে পারে, যেমন ব্যাকগ্রাউন্ডে ডেটা প্রসেসিং এবং UI আপডেট করা।
- সর্বোত্তম ব্যবহার: মাল্টি-কোর প্রসেসরের সম্পূর্ণ ব্যবহার করা যায়।

কনকারেন্সির উদাহরণ:

java
Copy code

```
class Task1 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task1: " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) { }
        }
    }
}

class Task2 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Task2: " + i);
            try { Thread.sleep(500); } catch (InterruptedException e) { }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Task1 t1 = new Task1();
        Task2 t2 = new Task2();

        t1.start();
        t2.start();
    }
}
```

- এখানে Task1 এবং Task2 দুইটি থ্রেড একসাথে চলবে, যেখানে একটি কাজ অন্যটি থামানোর আগে চলতে থাকবে। এটি কনকারেন্সি অর্জন করেছে, কিন্তু একসাথে (পারালাল) কাজ হচ্ছে না।

পার্থক্য: মুলতুবি থ্রেডিং এবং কনকারেন্সি

মুলতুবি থ্রেডিং	কনকারেন্সি
একাধিক থ্রেড একসাথে কাজ করার ক্ষমতা।	একাধিক কাজ একে অপরের সাথে সময় ভাগ করে কাজ করে।
একসাথে একাধিক কাজ চালানো যায়, বিশেষত মাল্টি-কোর প্রসেসর ব্যবহারের মাধ্যমে।	একাধিক কাজ একসাথে সম্পাদিত হয় না, তবে একে অপরকে বিভাজিত সময়ে সম্পাদন করা হয়।
এটি সাধারনত একটি প্রোগ্রামে একাধিক থ্রেড ব্যবহার করে।	এটি একটি সিস্টেমের মধ্যে একাধিক থ্রেড বা প্রসেস সমন্বয় করে।

পারালালিজম (Parallelism) এবং কনকারেন্সি:

- **পারালালিজম:** একাধিক কাজ একসাথে সম্পাদিত হয় (মাল্টি-কোর প্রসেসর ব্যবহার করে)।
- **কনকারেন্সি:** একাধিক কাজের মধ্যে সময় ভাগ করে কাজ করা, কিন্তু একসাথে সব কাজ একযোগে হয় না।

- **মুলতুবি থ্রেডিং:** এটি একাধিক থ্রেডের মাধ্যমে একাধিক কাজ সমান্তরালে (parallel) সম্পাদন করতে সক্ষম করে, যেটি অধিক পারফরম্যান্স দেয় এবং একাধিক কাজকে একসাথে করতে সাহায্য করে।
- **কনকারেন্সি:** এটি একাধিক কাজের মধ্যে সময় ভাগ করে কাজ সম্পাদন করে। একটি থ্রেড একে অপরের কাজের মধ্যে বিরতি নিয়ে সম্পাদন করে, তবে একসাথে সব কাজ হয় না।

এই দুটি ধারণা একসাথে জাভাতে থ্রেডিং এর শক্তিশালী ব্যবস্থাপনা সৃষ্টি করে, যা প্রোগ্রামিংয়ের কাজের গতি ও কার্যক্ষমতা উন্নত করে।

জাভা কালেকশন ফ্রেমওয়ার্ক (Java Collection Framework)

Java Collection Framework হলো একটি গ্রুপ যা বিভিন্ন ধরনের ডেটা স্ট্রাকচার এবং তাদের সাথে সম্পর্কিত ক্লাস, ইন্টারফেস এবং অ্যালগরিদমগুলির একটি সেট সরবরাহ করে। এটি জাভা প্রোগ্রামে ডেটা ম্যানিপুলেশন (যেমন, ডেটা ইনসার্ট, রিমুভ, সার্চ, সোর্ট ইত্যাদি) আরও সহজ এবং কার্যকরী করে তোলে।

কালেকশন (Collection)

একটি Collection হল এমন একটি অবজেক্ট যা একাধিক অবজেক্ট ধারণ করতে সক্ষম। এই অবজেক্টগুলো সাধারণত একই ধরনের ডেটা (যেমন, Integers, Strings) হতে পারে বা বিভিন্ন ধরনের ডেটাও হতে পারে (যেমন, Integers, Strings, Objects)।

জাভা কালেকশন ফ্রেমওয়ার্কে প্রধানত দুটি ধরনের কালেকশন ব্যবহৃত হয়:

- **Collection:** এটি সাধারণত একক (single) অবজেক্ট গোষ্ঠী নির্দেশ করে। এটি ইন্টারফেসের একটি বিস্তৃতি।
- **Map:** এটি key-value পেয়ার হিসাবে ডেটা ধারণ করে, যেখানে প্রতিটি key একটি মান (value) নির্দেশ করে।

জাভা কালেকশন ফ্রেমওয়ার্কের প্রধান উপাদান

- **Collection ইন্টারফেস**
- **List ইন্টারফেস**
- **Set ইন্টারফেস**
- **Queue ইন্টারফেস**
- **Map ইন্টারফেস**

১. Collection Interface

Collection ইন্টারফেস হল **List**, **Set** এবং **Queue** ইন্টারফেসের সুপার ইন্টারফেস। এটি সাধারণভাবে কালেকশন অপারেশনগুলির মৌলিক সেটের জন্য ব্যবহৃত হয়।

Collection Interface-এর কিছু গুরুত্বপূর্ণ মেথড:

- **add()**: একটি উপাদান যোগ করা।
- **remove()**: একটি উপাদান মুছে ফেলা।
- **size()**: কালেকশনের আকার নির্ণয় করা।
- **isEmpty()**: কালেকশনটি খালি কিনা তা যাচাই করা।
- **contains()**: নির্দিষ্ট উপাদানটি কালেকশনে রয়েছে কিনা যাচাই করা।

---

২. List Interface

List ইন্টারফেস হলো একটি **ordered collection** (অথবা সিকোয়েন্স) যা ডুপ্লিকেট উপাদান অনুমোদন করে এবং প্রতিটি উপাদানের একটি ইনডেক্স থাকে। এটি একটি **Indexed** ডেটা স্ট্রাকচার।

List-এর কিছু জনপ্রিয় ক্লাস:

- **ArrayList**: ডায়নামিক অ্যারে ভিত্তিক বাস্তবায়ন, দ্রুত অ্যাক্সেস।
- **LinkedList**: ডাবল লিঙ্কড লিস্ট ভিত্তিক বাস্তবায়ন, দ্রুত ইনসার্ট এবং ডিলিট।
- **Vector**: পুরনো ধরনের অ্যারে ভিত্তিক, তবে এর প্রপার্টি এখনকার ArrayList-এর মতো।

উদাহরণ:

java
Copy code

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        list.add("Apple");
        list.add("Banana");
```

```
        list.add("Orange");

        System.out.println(list);  // Output: [Apple, Banana, Orange]

        list.remove(1);  // Removes "Banana"
        System.out.println(list);  // Output: [Apple, Orange]

        System.out.println("Size: " + list.size());  // Output: Size: 2
    }
}
```

৩. Set Interface

Set ইন্টারফেস হলো একটি **unordered collection** (অথবা অর্ডারহীন সংগ্রহ) যেখানে **ডুপ্লিকেট উপাদান** অনুমোদিত নয়। Set এ একমাত্র একটি কপি থাকবে প্রতিটি উপাদানের। Set একাধিক উপাদান রাখতে পারে, কিন্তু তারা সব ইউনিক হতে হবে।

Set-এর কিছু জনপ্রিয় ক্লাস:

- **HashSet:** দ্রুত সঞ্চয় এবং খোঁজার জন্য ব্যবহার হয়, কিন্তু উপাদানগুলির কোন নির্দিষ্ট অর্ডার থাকে না।
- **LinkedHashSet:** HashSet এর মতো, তবে ইনসার্টের সময় এক্সেকিউটেড অর্ডার বজায় রাখে।
- **TreeSet:** উপাদানগুলিকে অর্ডার অনুসারে সজ্জিত রাখে (অথবা কমপ্যারেবল/কম্পেয়ারটার ব্যবহার করে অর্ডার করে)।

উদাহরণ:

java
Copy code

```
import java.util.*;

public class SetExample {
```

```
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();

        set.add("Apple");
        set.add("Banana");
        set.add("Apple");  // Duplicate entry, will be ignored

        System.out.println(set);  // Output: [Apple, Banana]
    }
}
```

---

8. Queue Interface

Queue ইন্টারফেস একটি **first-in, first-out** (FIFO) ডেটা স্ট্রাকচার। এটি একটি সংগ্রহ যা সাধারণত ব্যবহৃত হয় ডেটার সারি তৈরি করতে, যেমন প্রিন্টার কিউ বা থ্রেড সিডিউলিং। Queue একটি ডেটা স্ট্রাকচার যা উপাদানগুলিকে একে একে প্রক্রিয়া করে।

Queue-এর কিছু জনপ্রিয় ক্লাস:

- LinkedList: Queue ইন্টারফেস ইমপ্লিমেন্ট করে।
- PriorityQueue: এটি একটি কিউ, যেখানে উপাদানগুলিকে তাদের প্রাধান্য অনুসারে সারিবদ্ধ করা হয় (প্রথমে সবচেয়ে প্রাধান্যপ্রাপ্ত উপাদানটি প্রক্রিয়া করা হয়)।

উদাহরণ:

java
Copy code

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
```

```
        Queue<String> queue = new LinkedList<>();

        queue.add("Apple");
        queue.add("Banana");
        queue.add("Orange");

        System.out.println(queue);  // Output: [Apple, Banana, Orange]

        queue.remove();  // Removes "Apple"
        System.out.println(queue);  // Output: [Banana, Orange]

        System.out.println(queue.peek());  // Output: Banana (first element)
    }
}
```

---

৫. Map Interface

Map ইন্টারফেস একটি **key-value pair** সংগ্রহ করে। Map কোনো কালেকশন নয়, কারণ এটি Collection ইন্টারফেসের অধীনে আসে না। এটি ডেটাকে একটি নির্দিষ্ট key দিয়ে প্রাপ্ত করে এবং প্রতিটি key এর জন্য একটি নির্দিষ্ট value থাকে।

Map-এর কিছু জনপ্রিয় ক্লাস:

- **HashMap**: দ্রুত অ্যাক্সেস, কোন নির্দিষ্ট অর্ডার থাকে না।
- **LinkedHashMap**: ইনসার্টের সময় অর্ডার বজায় রাখে।
- **TreeMap**: key গুলি অর্ডার অনুযায়ী সাজানো থাকে (কমপ্যারেবল/কম্পেয়ারটার ব্যবহার করে)।

উদাহরণ:

java
Copy code

```
import java.util.*;
```

```
public class MapExample {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();

        map.put("Apple", "Fruit");
        map.put("Carrot", "Vegetable");
        map.put("Mango", "Fruit");

        System.out.println(map);  // Output: {Apple=Fruit, Carrot=Vegetable,
Mango=Fruit}

        System.out.println(map.get("Apple"));  // Output: Fruit
        map.remove("Carrot");
        System.out.println(map);  // Output: {Apple=Fruit, Mango=Fruit}
    }
}
```

Java Collection Framework-এর উদাহরণ

```
java
Copy code
import java.util.*;

public class JavaCollectionExample {
    public static void main(String[] args) {

        // List Example
        List<String> list = new ArrayList<>();
        list.add("One");
```

```
        list.add("Two");
        list.add("Three");
        System.out.println("List: " + list);

        // Set Example
        Set<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Apple");  // Duplicate
        System.out.println("Set: " + set);

        // Queue Example
        Queue<Integer> queue = new LinkedList<>();
        queue.add(1);
        queue.add(2);
        queue.add(3);
        System.out.println("Queue: " + queue);

        // Map Example
        Map<String, Integer> map = new HashMap<>();
        map.put("John", 30);
        map.put("Alice", 25);
        map.put("Bob", 35);
        System.out.println("Map: " + map);
    }
}
```

Java Collection Framework-এর কিছু অতিরিক্ত ফিচার:

- **Generics:** Collection Frameworkে **Generics** ব্যবহার করা হয়, যা টাইপ নিরাপত্তা (type safety) নিশ্চিত করে। এটি আপনার কালেকশনে ডেটা টাইপ নির্ধারণ করে, যেমন List<String> যেখানে String টাইপের ডেটা থাকবে।

- **Iterator Interface:** এটি কালেকশনের উপাদানগুলো একে একে প্রক্রিয়া করতে ব্যবহৃত হয়। এটি বিভিন্ন ধরনের লুপিং অপারেশন (যেমন, for-each লুপ) প্রদান করে।

জাভা কালেকশন ফ্রেমওয়ার্ক একটি শক্তিশালী টুল যা বিভিন্ন ধরনের ডেটা স্ট্রাকচার প্রদান করে, যা ডেটা ম্যানিপুলেশন এবং অপটিমাইজেশনে সাহায্য করে। এটি **List, Set, Queue,** এবং **Map** সহ বিভিন্ন কালেকশন ব্যবহারের মাধ্যমে কোডিংকে সহজ এবং আরও কার্যকরী করে তোলে।