Method chaining explanation

give me the deep and detailed explaination of Method Chaining so that i can able to understand it well and do not have any type of doubt in this.

so give me the deep and detailed explaination

Let's jump straight in.

---

1. What is Method Chaining? (Basic Idea)

Method Chaining means calling **multiple methods in a single statement**, one after another, using dots (`.`), like:

```
java

obj.method1().method2().method3();
```

Instead of writing:

```
java

obj.method1();
obj.method2();
obj.method3();
```

The **main condition** that makes chaining possible:

> Each method in the chain must return an object on which the **next** method can be called.

So the return type is very important.

---

2. Everyday Example (Before and After Chaining)

Without method chaining:

```
java

StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("Piyush");
sb.reverse();
```

```
System.out.println(sb);
```

With method chaining:

```
java

StringBuilder sb = new StringBuilder()
        .append("Hello")
        .append(" ")
        .append("Piyush")
        .reverse();

System.out.println(sb);
```

Both do the **same work**, but in the second code:

- `append()` returns the **same StringBuilder object**, so you can call `.append()` again.
- At the end, `.reverse()` also returns the same object.

So we can keep chaining.

---

3. How Does Method Chaining Work Internally?

Take this:

```
java

obj.a().b().c();
```

The JVM evaluates it **from left to right**:

1. `obj.a()` is executed
    - It returns some object (let's call it `x`).
2. Then on `x`, `.b()` is called: `x.b()`
    - It returns some object (let's call it `y`).
3. Then on `y`, `.c()` is called: `y.c()`
    - And so on...

So this:

```
java

obj.a().b().c();
```

Is conceptually like:

```
java

Type1 x = obj.a();
Type2 y = x.b();
```

```
Type3 z = y.c();  // final result
```

The **type returned by each method** decides what you can call next.

---

4. Types of Method Chaining

4.1 Chaining on the Same Object (Mutable & `return this`)

This is very common in Java.

Example: a `Person` class with setters.

Without chaining (void setters):

```
java

class Person {
    private String name;
    private int age;

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Usage:

```
java

Person p = new Person();
p.setName("Piyush");
p.setAge(21);
```

Here `setName` and `setAge` return `void`, so you **cannot** chain.

With chaining (`return this`):

```
java

class Person {
    private String name;
    private int age;

    // return the same object (this)
    public Person setName(String name) {
        this.name = name;
        return this;
    }

    public Person setAge(int age) {
        this.age = age;
        return this;
    }
}
```

Usage:

```
java

Person p = new Person()
        .setName("Piyush")
        .setAge(21);
```

Key idea:
`this` refers to the **current object**, so by returning `this`, we allow:

- The same object to be used again
- Multiple methods to be called on it in one statement

4.2 Chaining on New Objects (Immutable Objects)

Some classes don't modify the same object; instead, they create **new** objects on every method call.

Example: `String` in Java is immutable.

```
java

String s = "hello";

// A chain:
String result = s.toUpperCase().concat(" WORLD").trim();
```

Flow:

1. `s.toUpperCase()` → returns `"HELLO"` (new String object)
2. `"HELLO".concat(" WORLD")` → returns `"HELLO WORLD"` (new String)
3. `"HELLO WORLD".trim()` → returns `"HELLO WORLD"` (maybe same or new, but conceptually new)

So here, **each method returns a new object**, but we still chain because the return type has methods we can call.

4.3 Mixed Chaining (Different Types)

Sometimes each method returns a **different type**, still forming a chain.

Example: Streams

```
java

int sum = list.stream()
              .filter(n -> n % 2 == 0)
              .mapToInt(n -> n)
              .sum();
```

Types:

- `list.stream()` → returns `Stream<Integer>`

- `.filter(...)` → returns another `Stream<Integer>`
- `.mapToInt(...)` → returns `IntStream`
- `.sum()` → returns `int`

But this still works because **each return type** has the **next method**.

---

5. How to Implement Method Chaining in Your Own Class

Let's create a simple **StudentBuilder** style class.

5.1 Example: Simple Fluent Class

```
java

class Student {
    private String name;
    private int age;
    private String branch;

    // setters with method chaining
    public Student setName(String name) {
        this.name = name;
        return this; // important
    }

    public Student setAge(int age) {
        this.age = age;
        return this;
    }

    public Student setBranch(String branch) {
        this.branch = branch;
        return this;
    }

    public void show() {
        System.out.println("Name: " + name +
                           ", Age: " + age +
                           ", Branch: " + branch);
    }
}
```

Usage:

```
java

public class Main {
    public static void main(String[] args) {
        Student s = new Student()
                .setName("Piyush")
                .setAge(21)
                .setBranch("CSE");

        s.show();
    }
}
```

Why does this work?

- `new Student()` → returns a `Student` object

- `.setName(...)` → returns the **same** `Student` object (`this`)
- `.setAge(...)` → again returns `this`
- `.setBranch(...)` → again returns `this`

So it is exactly like:

```
java

Student temp = new Student();
temp = temp.setName("Piyush");
temp = temp.setAge(21);
temp = temp.setBranch("CSE");
```

But since each method returns the same `temp` object, chaining is natural.

---

6. Fluent Interface & Builder Pattern (Real Use Case)

Fluent Interface is a style of designing APIs so that you can write code almost like a sentence.

6.1 Builder Pattern with Method Chaining

Say we have a `User` class with many fields:

```
java

class User {
    private String username;
    private String email;
    private int age;

    // private constructor
    private User(String username, String email, int age) {
        this.username = username;
        this.email = email;
        this.age = age;
    }

    // Static nested builder class
    public static class Builder {
        private String username;
        private String email;
        private int age;

        public Builder setUsername(String username) {
            this.username = username;
            return this;
        }

        public Builder setEmail(String email) {
            this.email = email;
            return this;
        }

        public Builder setAge(int age) {
            this.age = age;
            return this;
        }

        public User build() {
            return new User(username, email, age);
        }
```

```
    }
}
```

Usage:

```
java

User user = new User.Builder()
        .setUsername("piyush_hazari")
        .setEmail("mepiyushhazari@gmail.com")
        .setAge(21)
        .build();
```

Flow:

- `new User.Builder()` → Builder object
- `.setUsername(...)` → returns same Builder (`this`)
- `.setEmail(...)` → same Builder
- `.setAge(...)` → same Builder
- `.build()` → returns final `User` object

This style is very common in real projects.

---

7. Method Chaining in Java Standard Library

7.1 `StringBuilder`

```
java

String result = new StringBuilder()
        .append("Hello ")
        .append("Piyush")
        .append("!")
        .toString();
```

- `append()` returns `StringBuilder` → chain more
- `toString()` returns `String` (end of chain)

7.2 Streams

```
java

List<Integer> list = List.of(1, 2, 3, 4, 5, 6);

int sumOfEvenSquares = list.stream()
        .filter(n -> n % 2 == 0)
        .map(n -> n * n)
        .reduce(0, (a, b) -> a + b);

System.out.println(sumOfEvenSquares);
```

Chaining happens because:

- `stream()` returns a Stream
- `filter()` returns a Stream
- `map()` returns a Stream
- `reduce()` returns a final value

7.3 `Optional`

```
java

Optional.of("hello")
        .map(String::toUpperCase)
        .ifPresent(System.out::println);
```

Every call returns `Optional` again (except `ifPresent`, which is terminal).

---

8. Advantages of Method Chaining

1. **Readable "flow" of operations**
   - Code can look like a pipeline of steps.
   - Easy to see the sequence at one place.
2. **Less boilerplate**
   - No need to create many temporary variables.
   - No repeated `obj.` lines.
3. **Great for configuration**
   - Builders and configuration objects are much cleaner with chaining:

     ```
     java

     config.setHost("localhost")
           .setPort(8080)
           .setSecure(true);
     ```

4. **Fluent APIs feel natural**
   - Code looks like a sentence: `query.from("users").where("age > 18").orderBy("name").`

---

9. Disadvantages / Things to Be Careful About

1. **Debugging long chains is harder**

   ```
   java

   list.stream()
       .filter(...)
       .map(...)
       .sorted(...)
   ```

```
    .limit(10)
    .collect(...);
```

If something goes wrong, it's not always obvious **which step** caused the issue.

Tip: you can break the chain:

```
java

Stream<Integer> s1 = list.stream();
Stream<Integer> s2 = s1.filter(...);
Stream<Integer> s3 = s2.map(...);
...
```

2. **NullPointerException in the middle of a chain**

```
java

obj.getA().getB().getC().doSomething();
```

If `obj.getA()` returns `null`, the rest will crash with `NullPointerException`.

Fix options:

- Null checks in between:

```
java

A a = obj.getA();
if (a != null) {
    B b = a.getB();
    ...
}
```

- Or use `Optional`:

```
java

Optional.ofNullable(obj.getA())
        .map(A::getB)
        .map(B::getC)
        .ifPresent(C::doSomething);
```

3. **Too much chaining reduces readability**

   One or two lines chaining is fine.
   10–15 methods in one chain might be **hard to read**.

4. **Side effects + chaining can be confusing**

   If your methods **change internal state** and also return `this`, it can be confusing if overused.
   Keep method behavior clear and predictable.

---

10. Summary in Simple Points

- Method chaining = calling methods one after another in a single statement using dots `.`:

```
java

obj.m1().m2().m3();
```

- It only works when each method returns:
  - Either the **same object** (`return this;`), or
  - A **new object** that has the next method.
- Common patterns:
  - `StringBuilder`, Streams, `Optional`, Builders, Fluent APIs.
- To implement in your own class:
  - Make your methods return the class itself (`YourClass`) instead of `void`.
  - Use `return this;` at the end.
- Benefits:
  - Clean, readable, less boilerplate, feels like a pipeline.
- Be careful:
  - Nulls, long chains (readability), debugging.

If you want, next I can:

- Take one **real-world style class** (like a `QueryBuilder` or `APIRequestBuilder`)
- And show you **step-by-step** how to design it with method chaining, with a full dry run of how the calls execute internally.