

Stream() in Java

1 What is stream() in Java?

stream() is a **Java 8 feature** used to **process collections** (List, Set, Map) in a **clean, readable, and functional way**.

Think of a **Stream as a pipeline**:

Data → Processing → Result

2 Why Streams were introduced (Java 8)?

Before Java 8:

- Code was **long**
- Needed **loops**
- Hard to read & maintain

Java 8 introduced:

- **Streams**
- **Lambda expressions**
- **Functional programming style**

Goal 🗒 **Less code, more clarity**

3 Without Stream (Old way)

Example: Convert CategoryEntity → CategoryResponse

```
List<CategoryResponse> responseList = new ArrayList<>();
```

```
for (CategoryEntity entity : categoryRepository.findAll()) {  
    CategoryResponse response = convertToResponse(entity);  
    responseList.add(response);  
}
```

```
return responseList;
```

- ✓ Works
 - ✗ More code
 - ✗ Manual loop
-

4 With Stream (Java 8 way)

```
return categoryRepository.findAll()  
    .stream()  
    .map(entity -> convertToResponse(entity))  
    .collect(Collectors.toList());
```

- ✓ Short
 - ✓ Clean
 - ✓ Easy to read
-

5 What exactly is stream()?

```
.stream()
```

- Converts a **Collection** into a **Stream**
- Does NOT store data
- Just **processes data one by one**

Example:

```
List<Integer> numbers = List.of(1, 2, 3, 4);  
numbers.stream();
```

Now Java can **process** these numbers.

6 Stream Pipeline Structure

Every stream has **3 parts**:

Source → Intermediate → Terminal

Example:

```
list.stream()  
  .map(...)  
  .collect(...)
```

Part	Meaning
stream()	Source
map()	Intermediate
collect()	Terminal

7 map() – Most Important Concept 🔥

What does map() do?

👉 **Transforms one object into another**

```
.map(entity -> convertToResponse(entity))
```

Means:

CategoryEntity → CategoryResponse

Simple Example:

```
List<Integer> numbers = List.of(1, 2, 3);
```

```
List<Integer> squares = numbers.stream()
```

```
.map(n -> n * n)
.collect(Collectors.toList());
```

Output:

```
[1, 4, 9]
```

8 filter() – Select specific data

```
list.stream()
    .filter(n -> n > 10)
    .collect(Collectors.toList());
```

☞ Keeps only values **greater than 10**

9 collect() – Final Result

collect() **ends the stream** and gives output.

```
.collect(Collectors.toList());
```

Other examples:

```
Collectors.toSet()
Collectors.toMap()
```

10 Full Real Spring Boot Example

Entity

```
class CategoryEntity {
    String name;
}
```

Response DTO

```
class CategoryResponse {  
    String name;  
}
```

Service

```
public List<CategoryResponse> read() {  
    return categoryRepository.findAll()  
        .stream()  
        .map(this::convertToResponse)  
        .collect(Collectors.toList());  
}
```

11 Why Streams are IMPORTANT in Spring Boot?

- ✓ Clean service layer
 - ✓ Easy DTO mapping
 - ✓ Less bugs
 - ✓ Industry standard
 - ✓ Used in **real projects**
-

13 Important Notes (Interview Ready)

- Stream **does not modify original data**
 - Stream is **used once**
 - Lazy execution (runs only on `collect()`)
 - **Stream = Modern way to process collections without loops**
-

filter() :

1 What is filter()?

filter() is a **Stream intermediate operation**.

☞ It **keeps only the elements that match a condition**

☞ Removes everything else

Simple meaning:

**“If condition is TRUE → keep it
If FALSE → discard it”**

2 Basic Syntax

`stream.filter(condition)`

Example:

```
.filter(n -> n > 10)
```

3 Very Simple Example (Numbers)

```
List<Integer> numbers = List.of(5, 12, 8, 20);
```

```
List<Integer> result = numbers.stream()  
    .filter(n -> n > 10)  
    .collect(Collectors.toList());
```

Step-by-step:

Number	Condition >10	Result
5	✗ false	removed
12	✓ true	kept
8	✗ false	removed
20	✓ true	kept

Output:

[12, 20]

4 filter() with Objects (REAL USE 🔥)

Entity

```
class CategoryEntity {  
    String name;  
    boolean active;  
}
```

Get only ACTIVE categories

```
List<CategoryEntity> activeCategories =  
    categoryRepository.findAll()  
        .stream()  
        .filter(cat -> cat.isActive())  
        .collect(Collectors.toList());
```

👉 Only active == true records returned

5 filter() + map() together (Most common)

```
return categoryRepository.findAll()  
    .stream()  
    .filter(cat -> cat.isActive())
```

```
.map(this::convertToResponse)  
.collect(Collectors.toList());
```

Flow:

DB Data

- filter active
 - convert to DTO
 - return list
-

6 Multiple Conditions in filter()

Using AND (&&)

```
.filter(c -> c.isActive() && c.getName().startsWith("E"))
```

Using OR (||)

```
.filter(c -> c.isActive() || c.getName().equals("Food"))
```

7 filter() with String conditions

```
.filter(name -> name.contains("phone"))  
.filter(name -> name.equalsIgnoreCase("electronics"))
```

8 Without filter() (Old way)

```
List<CategoryEntity> result = new ArrayList<>();  
  
for (CategoryEntity c : categoryRepository.findAll()) {  
    if (c.isActive()) {  
        result.add(c);  
    }  
}
```



```
}
```

9 With filter() (Modern way)

```
categoryRepository.findAll()  
    .stream()  
    .filter(CategoryEntity::isActive)  
    .collect(Collectors.toList());
```

- ✓ Less code
 - ✓ More readable
 - ✓ Standard practice
-

10 Important Rules (Must Know)

- filter() **does not change original list**
 - filter() returns a **new stream**
 - It runs **only when terminal operation is called**
 - filter() uses **Predicate** (boolean return)
-

11 Real Project Scenario (Security)

✗ BAD (return everything)

```
return userRepository.findAll();
```

✓ GOOD (filter sensitive data)

```
return userRepository.findAll()  
    .stream()  
    .filter(User::isActive)  
    .map(this::toDTO)  
    .collect(Collectors.toList());
```

12 One-line Summary (Interview)

filter() selects data based on condition in streams

Map()

1 What is map()?

map() is a **Stream intermediate operation**.

- ☞ It **transforms one object into another**
- ☞ **Size stays same**, only **shape/value changes**

Simple meaning:

Input → Converted Output

2 Basic Syntax

```
stream.map(element -> newElement)
```

The function inside map() **must return something**.

3 Very Simple Example (Numbers)

```
List<Integer> numbers = List.of(1, 2, 3, 4);
```

```
List<Integer> squares = numbers.stream()  
    .map(n -> n * n)  
    .collect(Collectors.toList());
```

Step-by-step:

Input map logic Output

1	1 * 1	1
2	2 * 2	4
3	3 * 3	9
4	4 * 4	16

Output:

[1, 4, 9, 16]

4 map() with Strings

```
List<String> names = List.of("java", "spring");
```

```
List<String> upper = names.stream()
    .map(name -> name.toUpperCase())
    .collect(Collectors.toList());
```

Output:

[JAVA, SPRING]

5 map() with Objects (REAL USE 🔥)

Entity

```
class CategoryEntity {
    String name;
    String description;
}
```

DTO

```
class CategoryResponse {  
    String name;  
}
```

Convert Entity → DTO using map()

```
return categoryRepository.findAll()  
    .stream()  
    .map(entity -> {  
        CategoryResponse res = new CategoryResponse();  
        res.setName(entity.getName());  
        return res;  
    })  
    .collect(Collectors.toList());
```

☞ Each CategoryEntity becomes CategoryResponse

6 Cleaner way (Method Reference)

```
.map(this::convertToResponse)
```

Same as:

```
.map(entity -> convertToResponse(entity))
```

7 map() + filter() (Most common combo)

```
return categoryRepository.findAll()  
    .stream()  
    .filter(CategoryEntity::isActive)  
    .map(this::convertToResponse)  
    .collect(Collectors.toList());
```

Flow:

DB → filter → map → response

8 map() vs filter() (Very Important)

Feature	map()	filter()
Purpose	Transform	Select
Changes size?	✗ No	✓ Yes
Returns	New object	Same object
Example	Entity → DTO	active == true

9 Without map() (Old way)

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

for (CategoryEntity e : categoryRepository.findAll()) {
    list.add(convertToResponse(e));
}
```

11 With map() (Modern way)

```
categoryRepository.findAll()
    .stream()
    .map(this::convertToResponse)
    .collect(Collectors.toList());
```

- ✓ Clean
 - ✓ Standard
 - ✓ Used in real projects
-

11 Common Mistakes ✖

✖ Not returning value

```
.map(e -> {  
    e.setName("test"); // wrong  
})
```

✔ Correct

```
.map(e -> {  
    e.setName("test");  
    return e;  
})
```

12 Interview One-Line Answer

`map()` transforms each element of a stream into another form

13 Real Spring Boot Use

- Entity → DTO
 - DTO → Entity
 - Modify values
 - Hide sensitive fields
-

Here is **VERY SIMPLE Java code** (no Spring, no DB) to understand **stream, filter, map**.

Example 1 Without Stream (Normal Java)

```
import java.util.*;

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

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

        for (int n : numbers) {
            if (n > 2) {
                result.add(n * 2);
            }
        }

        System.out.println(result);
    }
}
```

Output

[6, 8, 10]

Example 2 With Stream (Java 8)

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

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

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



```
List<Integer> result = numbers.stream()
    .filter(n -> n > 2)
    .map(n -> n * 2)
    .collect(Collectors.toList());

System.out.println(result);
}
}
```

Output

[6, 8, 10]

How Stream Works (Step-by-step)

numbers = [1,2,3,4,5]

```
stream()
→ filter(n > 2) → [3,4,5]
→ map(n * 2) → [6,8,10]
→ collect(List) → result
```

Example 3 String Example (Very Easy)

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

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

        List<String> names = Arrays.asList("java", "spring", "boot");

        List<String> result = names.stream()
            .map(name -> name.toUpperCase())
            .collect(Collectors.toList());
```

```
        System.out.println(result);  
    }  
}
```

Output

[JAVA, SPRING, BOOT]

One-line Summary

- `filter()` → selects data
 - `map()` → transforms data
 - `collect()` → gives final result
-

Below is a **COMPLETE, READY-TO-RUN Spring Boot project**
(Simple, minimal, beginner-friendly, uses **Stream for Entity** → **DTO, hide fields**)

You can **copy-paste and run**.

✓ Project: User Management (Spring Boot + Stream)

Tech

- Java 17+
 - Spring Boot
 - JPA
 - H2 DB (in-memory)
-

1 pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>stream-demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.1</version>
  </parent>

  <properties>
    <java.version>17</java.version>
```

```
</properties>

<dependencies>
  <!-- Web -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <!-- JPA -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>

  <!-- H2 DB -->
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
</project>
```

2 application.properties

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
```

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2
```

3 Main Class

```
@SpringBootApplication
public class StreamDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(StreamDemoApplication.class, args);
    }
}
```

4 Entity (DB Object)

```
@Entity
public class UserEntity {

    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String email;
    private String password; // sensitive

    // getters & setters
}
```

5 DTOs

Request DTO

```
public class UserRequest {
    public String name;
    public String email;
    public String password;
}
```

```
}
```

Response DTO (NO password)

```
public class UserResponse {  
    public Long id;  
    public String name;  
    public String email;  
}
```

6 Repository

```
public interface UserRepository  
    extends JpaRepository<UserEntity, Long> {  
}
```

7 Service (🔥 STREAM USED HERE)

```
@Service  
public class UserService {  
  
    private final UserRepository repo;  
  
    public UserService(UserRepository repo) {  
        this.repo = repo;  
    }  
  
    // DTO → Entity  
    public void save(UserRequest req) {  
        UserEntity user = new UserEntity();  
        user.setName(req.name);  
        user.setEmail(req.email);  
        user.setPassword(req.password);  
        repo.save(user);  
    }  
}
```

```
// Entity → DTO (Stream)
public List<UserResponse> getAll() {

    return repo.findAll()
        .stream()
        .map(this::toResponse)
        .collect(Collectors.toList());
}

private UserResponse toResponse(UserEntity user) {
    UserResponse dto = new UserResponse();
    dto.id = user.getId();
    dto.name = user.getName();
    dto.email = user.getEmail();
    return dto;
}
}
```

8 Controller

```
@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService service;

    public UserController(UserService service) {
        this.service = service;
    }

    @PostMapping
    public String create(@RequestBody UserRequest req) {
        service.save(req);
        return "User created";
    }

    @GetMapping
    public List<UserResponse> getAll() {
```

```
        return service.getAll();
    }
}
```

9 Test API (Postman / Browser)

+ Create User

POST http://localhost:8080/users

```
{
  "name": "Aniket",
  "email": "aniket@gmail.com",
  "password": "secret123"
}
```

⬇ Get Users

GET http://localhost:8080/users

```
[
  {
    "id": 1,
    "name": "Aniket",
    "email": "aniket@gmail.com"
  }
]
```

✓ password hidden

✓ Entity → DTO using Stream

🔄 What you learned (REAL PROJECT)

- Entity → DTO using map()
- DTO → Entity
- Hide sensitive fields

- Clean service layer
 - Industry-standard design
-

Interview One-Line

Streams are used in Spring Boot to transform entities into DTOs, apply business logic, and return secure, clean API responses.

Next (tell me one):

- same project **WITHOUT** stream
 - add **filter / active users**
 - add **update & delete**
 - connect **React frontend**
-

Perfect 👍

Below is **ADD + UPDATE + DELETE** added to the **same ready project**.

🔗 1 Update API (PUT)

Service – UPDATE

```
public UserResponse update(Long id, UserRequest req) {  
  
    UserEntity user = repo.findById(id)  
        .orElseThrow(() -> new RuntimeException("User not found"));  
  
    user.setName(req.name);  
    user.setEmail(req.email);  
}
```

```
user.setPassword(req.password);

UserEntity updated = repo.save(user);
return toResponse(updated);
}
```

Controller – UPDATE

```
@PutMapping("/{id}")
public UserResponse update(
    @PathVariable Long id,
    @RequestBody UserRequest req) {

    return service.update(id, req);
}
```

Test (Postman)

```
PUT http://localhost:8080/users/1
{
  "name": "Aniket Updated",
  "email": "aniket.updated@gmail.com",
  "password": "newpass"
}
```

🔹 2 Delete API (DELETE)

Service – DELETE

```
public void delete(Long id) {

    UserEntity user = repo.findById(id)
        .orElseThrow(() -> new RuntimeException("User not found"));

    repo.delete(user);
}
```

Controller – DELETE

```
@DeleteMapping("/{id}")
public String delete(@PathVariable Long id) {

    service.delete(id);
    return "User deleted successfully";
}
```

Test

DELETE http://localhost:8080/users/1

🔗 3 Get User by ID (Optional but REAL)

Service

```
public UserResponse getById(Long id) {

    return repo.findById(id)
        .map(this::toResponse) // STREAM STYLE
        .orElseThrow(() -> new RuntimeException("User not found"));
}
```

Controller

```
@GetMapping("/{id}")
public UserResponse getById(@PathVariable Long id) {
    return service.getById(id);
}
```

🔄 Final API List (CRUD)

Method	URL	Action
POST	/users	Create
GET	/users	Get all
GET	/users/{id}	Get by id
PUT	/users/{id}	Update
DELETE	/users/{id}	Delete

💧 Where Stream is Used (IMPORTANT)

- `findAll().stream().map(Entity → DTO)`
 - `Optional.map()` for `getById`
 - Sensitive fields hidden (password)
-

🎯 Interview Ready Line

Streams are used in service layer for clean entity-to-DTO mapping, while update and delete operations use repository methods with proper validation.
