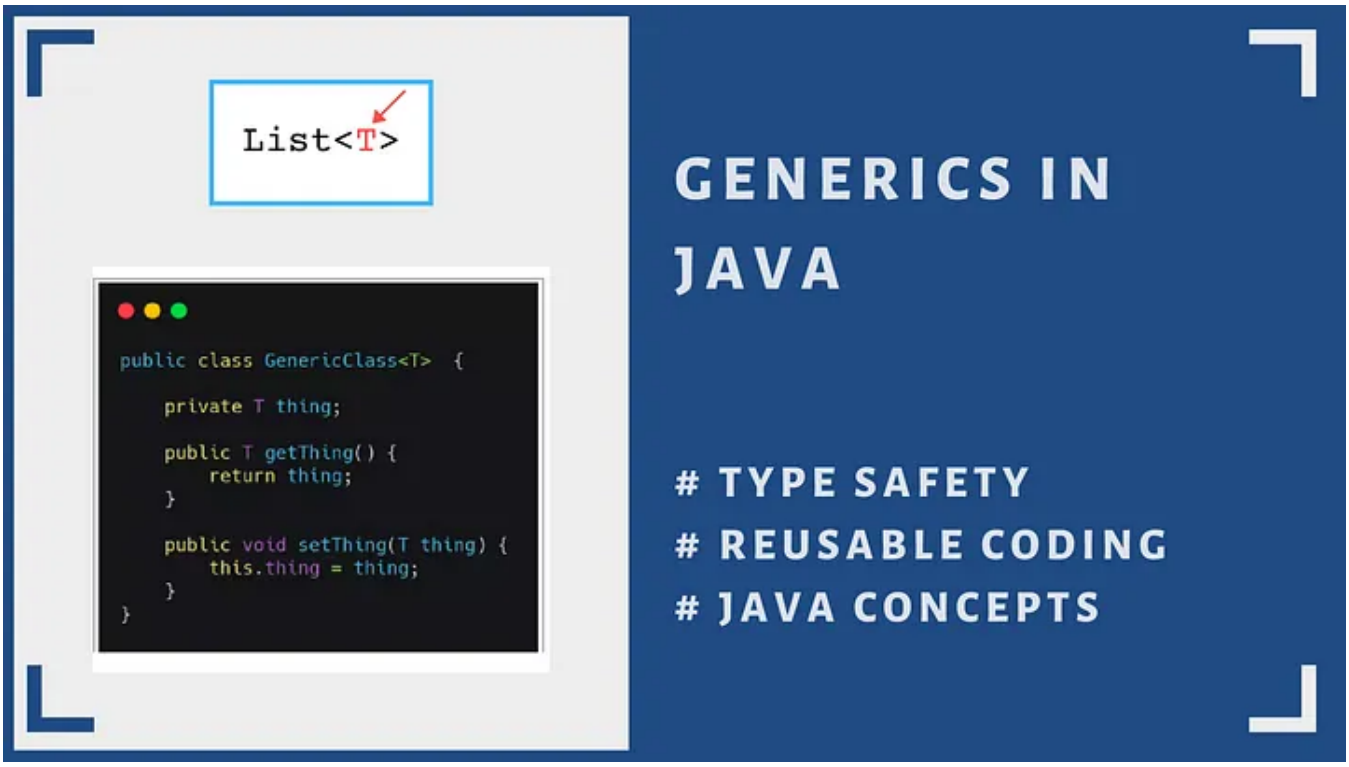
Generics in Java

Salitha Chathuranga · Follow
6 min read · Apr 30, 2023

Listen Share

It's all about type safety



Hi all!!!

I thought of writing a widely used but less discussed topic in Java. That is **Generics**! We use it, but majority of the developers don't know about it as I have experienced.

Let me clear this…have you ever used List or **ArrayList** in Java? Most probably, answer should be YES. Right? Without collections, we can't even think of an easy way of handling data. So, do you remember how we define an ArrayList?

```
List<Integer> numbers = new ArrayList<>(); // with Generics
```

This is the way we declare it. So, we have used generics. 😃 Here, *<Integer>* is the Generic we passed. That is a Type. After we create list like this, you can **only add integers** to the list.

You may remember, if we define the list like below, we would be able to add any type of data which is extended from Object super class, to the list.

```
List numbers = new ArrayList(); // without Generics
```

We can achieve Type Safety for this List, after we add generics.

Generics means **parameterized types.** Java let us to create a single class, interface, and method that can be used with different types of data(objects) within the Generics domain.

Advantages in Generics would be:

- Code Reusability — we can use a common code with multiple object types
- Compile-time Type Checking — Java will check the generics code at the compile time against errors
- Type Safety — we can restrict adding unnecessary data
- Usage in Collections — Collections need object types to deal with data

Let's take and example to explain why we need Generics..

Imagine you have to print Numbers and Texts using a printer class. Printer has a method that accepts the data while creating it.

In traditional way, we will have to creat 2 classes since we have 2 types of data: Number(Integer) and Text(String).

```
public class TextPrinter {
    private final String data;

    public TextPrinter(String data) {
        this.data = data;
    }

    public void print() {
        System.out.println("print::: " + data);
    }
}
```

```
public class NumberPrinter {
    private final Integer data;

    public NumberPrinter(Integer data) {
        this.data = data;
    }

    public void print() {
        System.out.println("print::: " + data);
    }
}
```

How to use:

```
public class GenericsMain {
    public static void main(String[] args) {
        NumberPrinter numberPrinter = new NumberPrinter(5);
        numberPrinter.print(); // output = print::: 5
        TextPrinter textPrinter = new TextPrinter("Hello");
        textPrinter.print();   // output = print::: Hello
    }
}
```

You can see we have **code duplication**! Data type is the only difference here!

We can simply use a Printer with a Generic here. Then we will only have 1 Printer! 😎

Let's deep dive into Generics and see how we achieve this... 😎

Create a Generic

I'm taking the above simple example and will show how to create a **Generic** Printer.

```
public class Printer<T> {
    private final T data;

    public Printer(T data) {
        this.data = data;
    }

    public void print() {
        System.out.println("print::: " + data);
    }
}
```

How to use:

```
Printer<Integer> integerPrinter = new Printer<>(5);
integerPrinter.print();   // output = print::: 5

Printer<String> stringPrinter = new Printer<>("Hello");
stringPrinter.print();   // output = print::: Hello

Printer<Double> doublePrinter = new Printer<>(45.34);
doublePrinter.print();   // output = print::: 45.34

Printer<Long> longPrinter = new Printer<>(5L);
longPrinter.print();z    // output = print::: 5
```

Now we only have 1 class! It accepts a Type. Here, T is used to denote the Type as a common standard. We can even create printer objects for other data types also like Double/Long. **Code reusability** is achieved in style. 😎

We can create Generic classes which accepts more than 1 type. Look at the below example. It accepts an Integer and a String both.

```
public class MultiPrinter<T, V> {
    private final T data1;
    private final V data2;

    public MultiPrinter(T data1, V data2) {
        this.data1 = data1;
        this.data2 = data2;
    }

    public void print() {
        System.out.println("print::: " + data1 + " : " + data2);
    }
}
```

```
MultiPrinter<Integer, String> multiPrinter = new MultiPrinter<>(5, "Hello");
multiPrinter.print(); // output = print::: 5 : Hello
```

Java Type Naming conventions

- E — Element (used in Collections)
- K — Key (Used in Map)
- N — Number
- T — Type
- V — Value (Used in Map)
- S, U, V etc. — 2nd, 3rd, 4th types

Bounded Generics

This is an advanced version of Generics. We can restrict more and achieve **more type safety** with bounded Generics.

Let's say we have an **AnimalPrinter** class which can only print animal details. No other objects are allowed to be used with it. How to achieve this?

```
public class Animal {
    private final String name;
    private final String color;
    private final Integer age;

    public Animal(String name, String color, Integer age) {
        this.name = name;
        this.color = color;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getColor() {
        return color;
    }

    public Integer getAge() {
        return age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Animal animal = (Animal) o;
        return Objects.equals(name, animal.name) && Objects.equals(color, anima
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, color, age);
    }
}

public class Cat extends Animal {
    public Cat(String name, String color, Integer age) {
        super(name, color, age);
    }
}

public class Dog extends Animal {
    public Dog(String name, String color, Integer age) {
        super(name, color, age);
```

```
    }
}
```

```
public class AnimalPrinter<T extends Animal> {
    private final T animalData;

    public AnimalPrinter(T animalData) {
        this.animalData = animalData;
    }

    public void print() {
        System.out.println("Name::: " + animalData.getName());
        System.out.println("Color::: " + animalData.getColor());
        System.out.println("Age::: " + animalData.getAge());
    }
}
```

In this class, T extends Animal part does the job! We have limited our generic for Dog and Cat!

```
AnimalPrinter<Cat> animalPrinter1 = new AnimalPrinter<>(new Cat("Jim", "brown",
animalPrinter1.print();
AnimalPrinter<Dog> animalPrinter2 = new AnimalPrinter<>(new Dog("Rocky", "black
animalPrinter2.print();
```

If we try to define the printer with another Object type, compiler will complain like this => *"Type parameter 'java.lang.Object' is not within its bound; should extend 'generics.Animal'*

Multiple Bounds

Let's say we want to add some more features to the Printer generic. We can achieve it like this.

```
public class AnimalPrinter<T extends Animal & Serializable> {
    ..................
}
```

I have provided Serializable functionality using Serializable interface. There are important things to remember here.

- We must implement interface in our child classes(Cat and Dog).
- Class should come first and the & and interface.
- Only 1 class can be extended since Java does not support multiple inheritance.

Wildcards With Generics

Wildcards are represented by the question mark *?* in Java, and we use them to refer to an unknown type. This can be used as a parameter type with Generics. Then it will accept any type. I have used a List of any object as a method argument using wild card, in the below code.

```
public static void printList(List<?> list) {
    System.out.println(list);
}

printList(
    Arrays.asList(
        new Cat("Jim", "brown", 2),
        new Dog("Rocky", "black", 5)
    )
);
printList(Arrays.asList(50, 60));
printList(Arrays.asList(50.45, 60.78));

// output:
// [generics.Cat@b1fa3959, generics.Dog@62294cd9]
// [50, 60]
// [50.45, 60.78]
```

List can be of any type now!!!

1 Upper Bounded Wild Cards

Consider this example:

```
public static void printAnimals(List<Animal> animals) {
      animals.forEach(Animal::eat);
}
```

If we imagine a subtype of *Animal*, such as a *Dog*, we can't use this method with a list of *Dog*, even though *Dog* is a subtype of *Animal*. We can do this with a wild card.

```
public static void printAnimals(List<? extends Animal> animals) {
    ...
}
```

Now this method works with type ***Animal*** and all ***its subtypes***.

```
printAnimals(
    Arrays.asList(
        new Cat("Jim", "brown", 2),
        new Dog("Rocky", "black", 5)
    )
);
```

This is called an **upper-bounded wildcard**, where type ***Animal*** is the upper bound.

2 Lower Bounded Wild Cards

We can also specify wildcards with a lower bound, where the unknown type has to be a **super type of the specified type**. Lower bounds can be specified using the ***super*** keyword followed by the specific type.

Example:

```
public static void addIntegers(List<? super Integer> list){
    list.add(new Integer(70));
}
```

Generic Methods

Imagine we need a method which takes different data types and do something. We


```
public static <T> void call(T data) {
    System.out.println(data);
}

call("hello");
call(45);
call(15.67);
call(5L);
call(new Dog("Rocky", "black", 5));

/* output:
    hello
    45
    15.67
    5
    generics.Dog@62294cd9
*/
```

If we want to return data instead of VOID, we can do that also.

```
public static <T> T getData(T data) {
    return data;
}

System.out.println(getData("Test"));   // output: Test
```

We can accept multiple data types also in a generic method.

```
public static <T, V> void getMultiData(T data1, V data2) {
    System.out.println("data 1: " + data1);
    System.out.println("data 2: " + data2);
}

getMultiData(50, "Shades of Grey");
```

I think I have covered almost all the things to be learnt in Generics. So, this would be an ideal article for you to practice Generics in Java. ❤️

I will bring you another Java stuff next time.

Bye guys! 🙌

Java Generics Advanced Java

Follow

Written by Salitha Chathuranga

1.2K Followers

Senior Software Engineer at Sysco LABS | Senior Java Developer | Blogger

More from Salitha Chathuranga

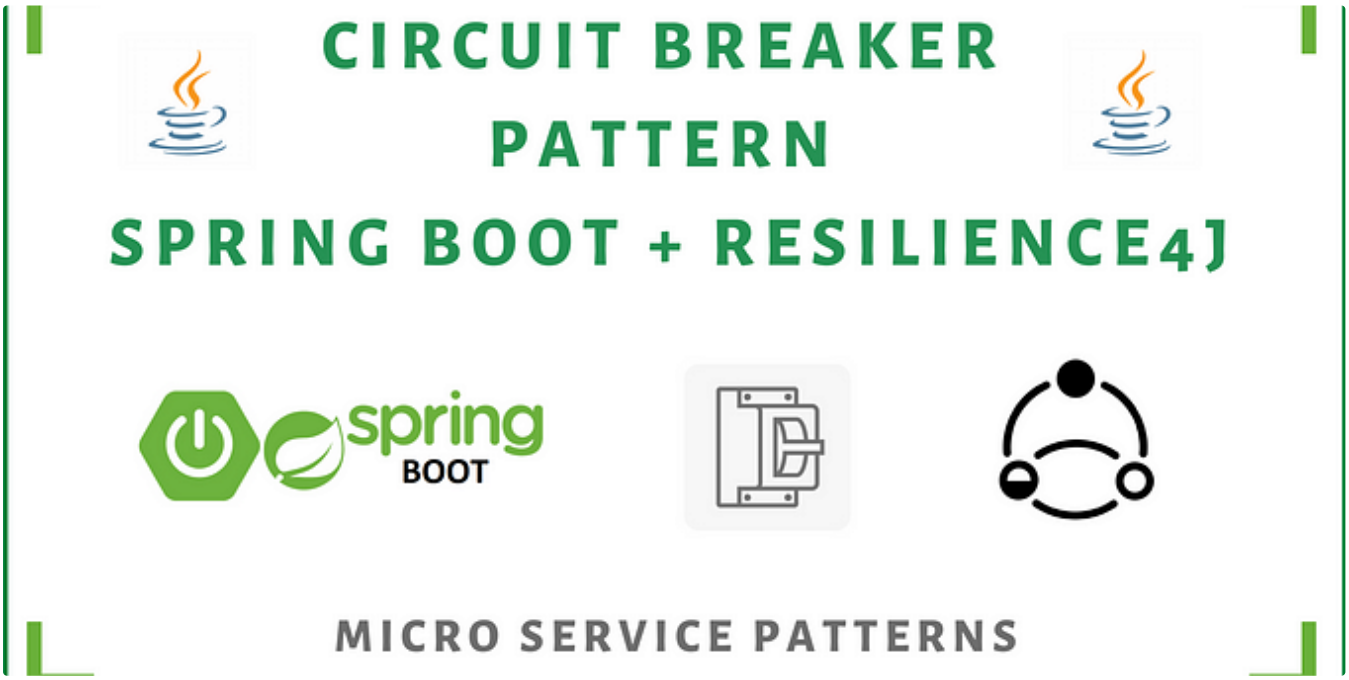
Salitha Chathuranga

Validation and Exception Handling in Spring Boot

Let’s validate incoming requests in our APIs

8 min read · Jul 19, 2022

461 3



Salitha Chathuranga

Micro Service Patterns: Circuit Breaker with Spring Boot

Let’s learn micro service design patterns

11 min read · Sep 17, 2022

95 4

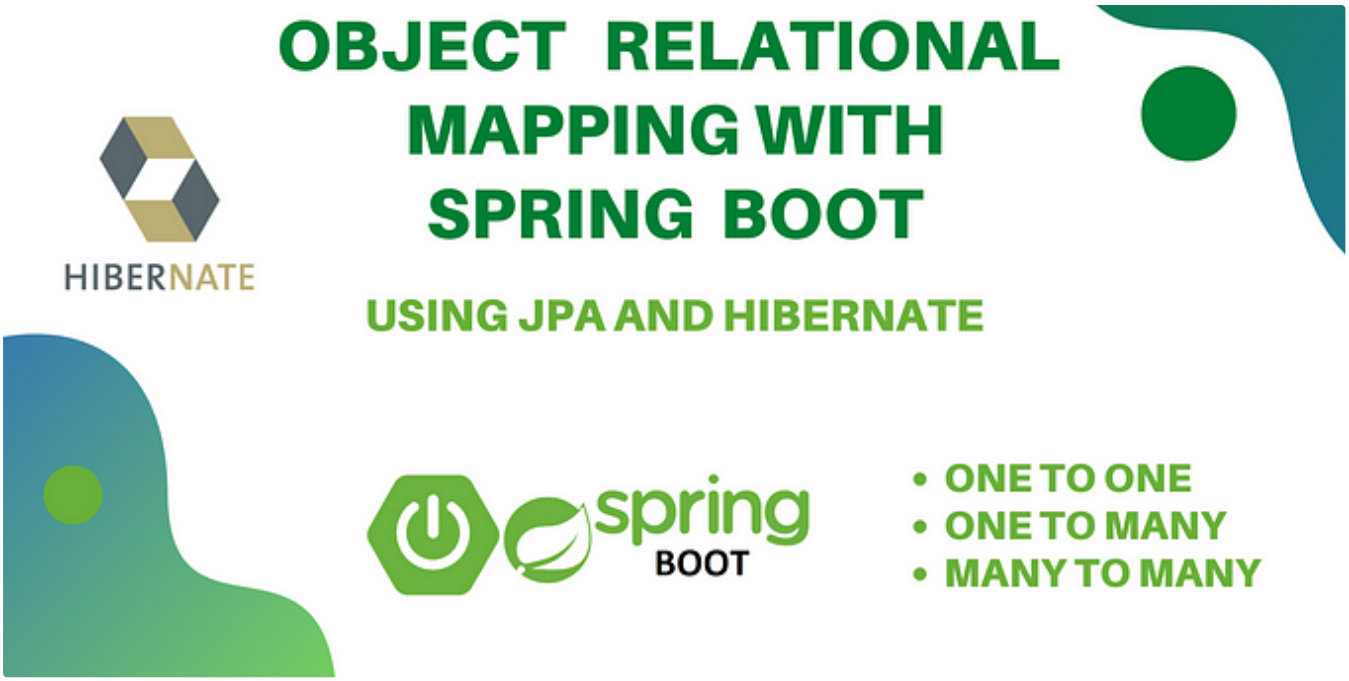
Salitha Chathuranga

Unit and Integration Testing in Spring Boot Micro Service

Let’s write tests using Mockito and JUnit

15 min read · Sep 12, 2022

96 2



Salitha Chathuranga

Object Relational Mapping with Spring Boot, JPA and Hibernate

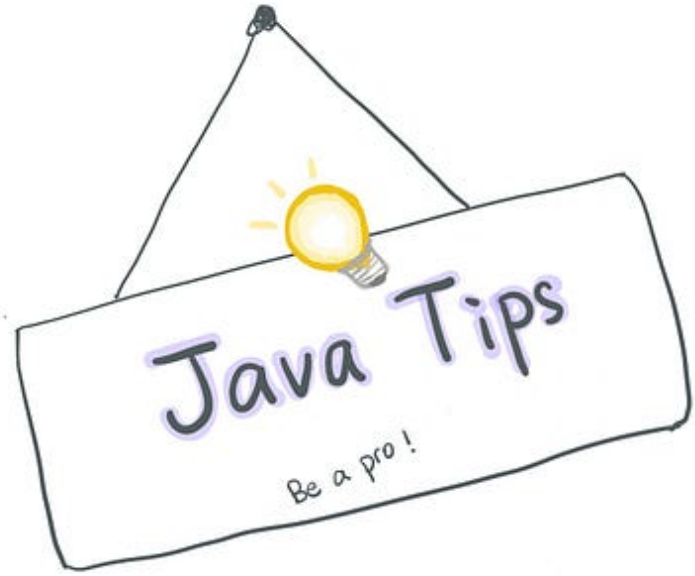
Let’s deal with object relationships

7 min read · Sep 20, 2022

88 3

See all from Salitha Chathuranga

Recommended from Medium



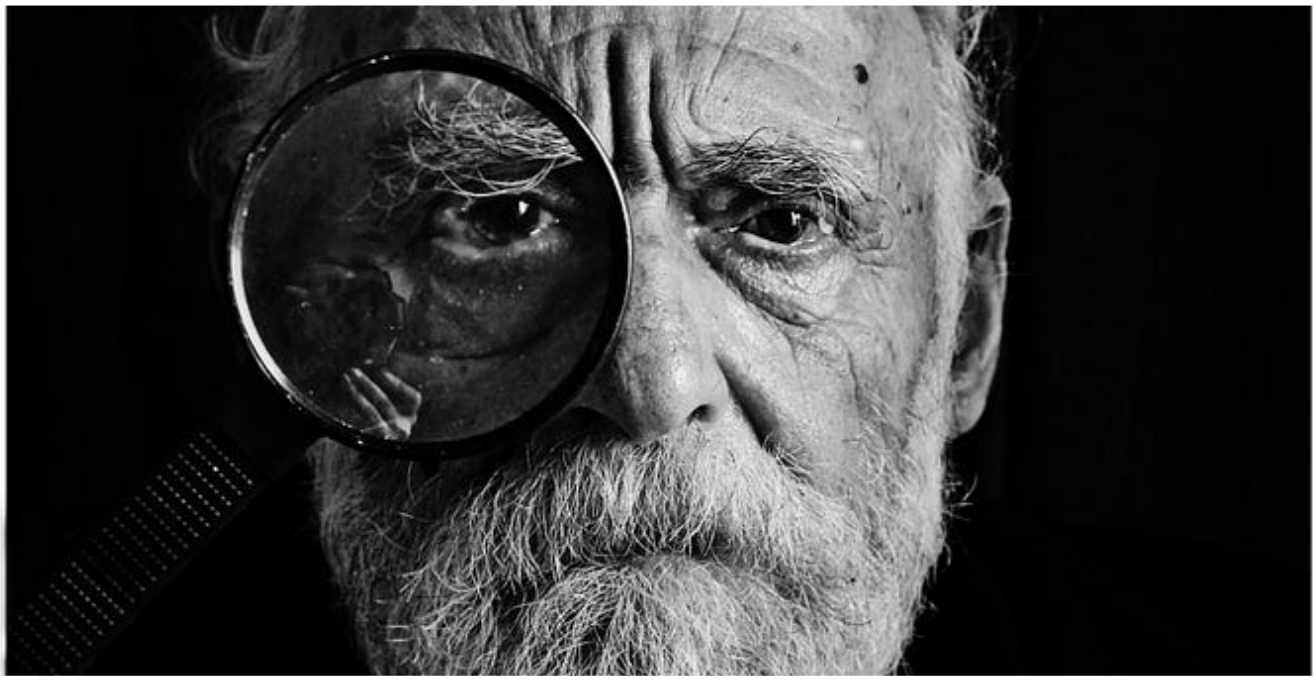
Jingnu An

A Tip To Start Using Java HashMap Like A Pro

Take a look at the following hash map. We are going to use it as an example to unlock the pro tip.

4 min read · Aug 4, 2023

313 4



Himani Prasad

Validations in Spring Boot

Validation is like a quality check for data. Just like a teacher marks your answers right or wrong, validation checks if the information...

6 min read · Aug 23, 2023

109 3

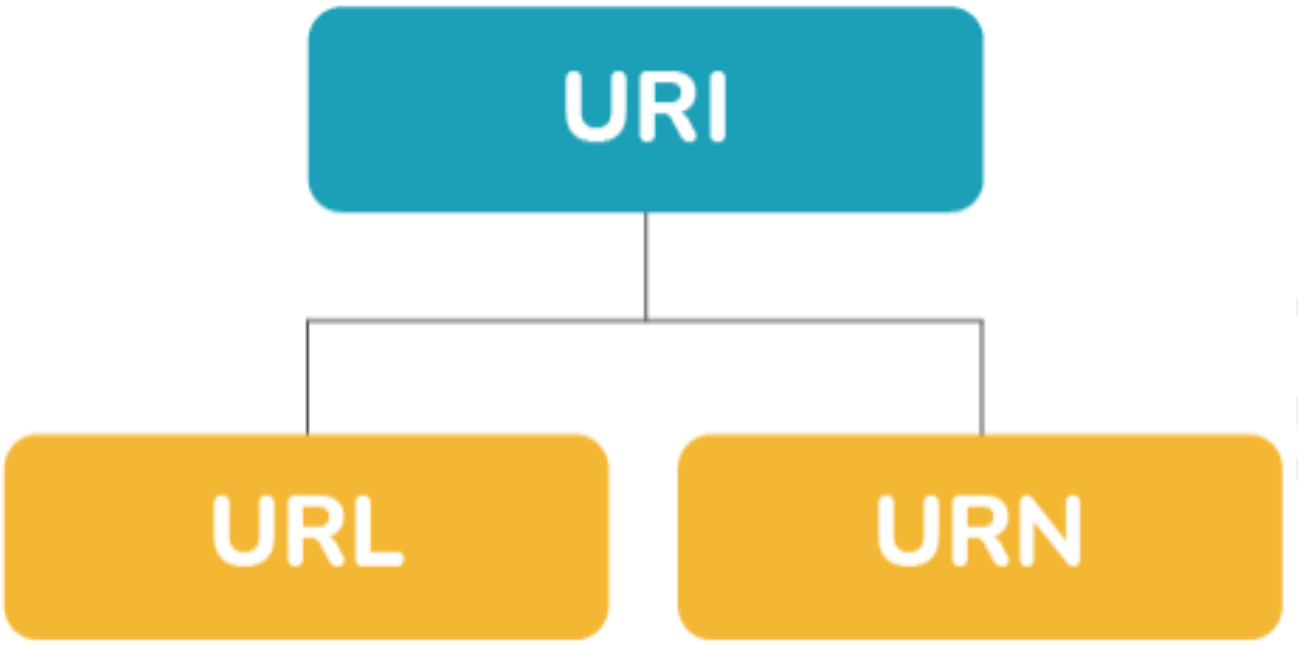
Lists

General Coding Knowledge

20 stories · 742 saves

data science and AI

38 stories · 32 saves



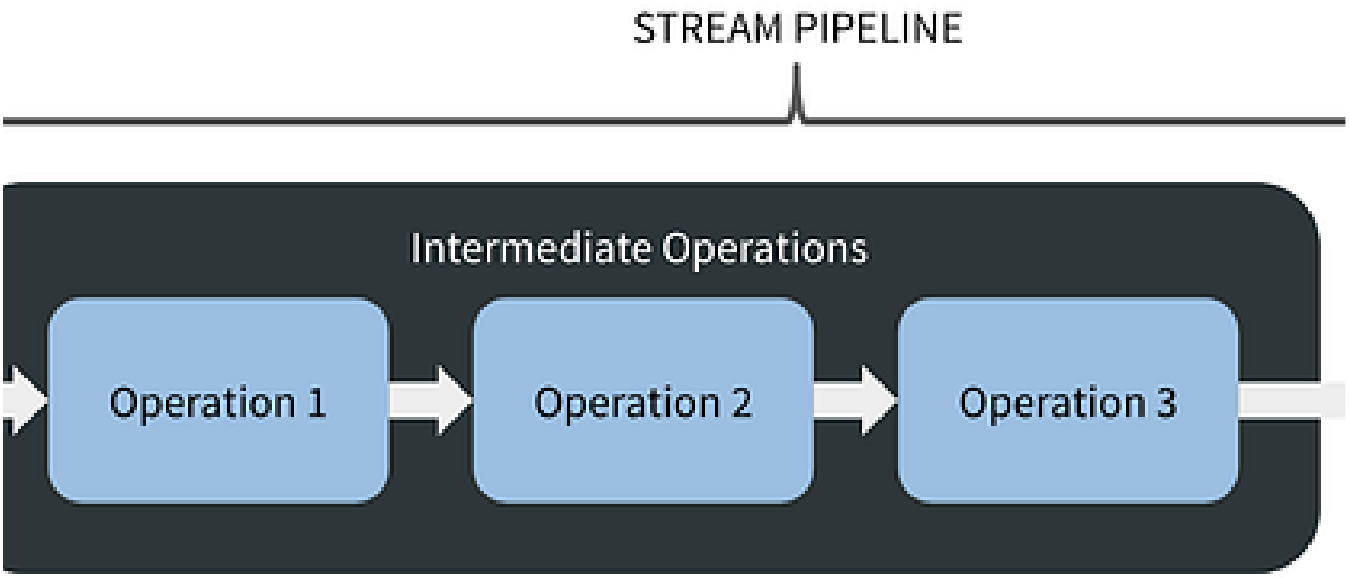
Abhishek Singh in Javarevisited

10 REST API Basic Interview Questions

1- What do you understand by RESTful Web Services ?

6 min read · Jul 17, 2023

334 3



Amirhosein Gharaati in Stackademic

Learn Java Stream API Practically

Streams was one of the major features added to Java 8. This tutorial is an introduction to the many functionalities supported by streams...

10 min read · Aug 15, 2023

305 3

Bubu Tripathy

Best Practices: Entity Class Design with JPA and Spring Boot

In the world of modern software development, efficient design and implementation of entity classes play a crucial role in building robust...

8 min read · Aug 10, 2023

480 7

Sharad Pawar in Towards Dev

Arrays.asList() vs List.of() in java

What is the difference between Arrays.asList() and List.of() in Java?

3 min read · Aug 26, 2023

237 5

See more recommendations